SERIE B – INFORMATIK

Sprachen für parallele objektorientierte Programmierung

Alexandra Weidmann

Bericht B 95-09
Mai 1995

Zusammenfassung: In letzter Zeit wurden eine ganze Reihe von objektorientierten Sprachen zur parallelen Programmierung entworfen und implementiert. Einige dieser Sprachen werden hier einander gegenübergestellt. Das Hauptaugenmerk der Arbeit liegt einerseits auf den bereitgestellten Konzepten zur Bewältigung der Komplexität, die sich durch die Parallelisierung ergibt, und andererseits auf der Flexibilisierung von Synchronisation und Kommunikation zur Optimierung der Parallelisierbarkeit von Programmausführungen.

Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin
Takustr. 9
D-14195 Berlin
weidmann@inf.fu-berlin.de
http://www.inf.fu-berlin.de

Inhaltsverzeichnis

1 Einleitung

Durch Parallelität[1] bzw. Nebenläufigkeit[2] erhöht sich die Komplexität der Programmierung zum einen durch die Notwendigkeit der Zerteilung und Verteilung von Daten und Kontrollflüssen und zum anderen durch die sich daraus ergebende Notwendigkeit zur Synchronisation. Hinzu kommt bei Architekturen mit verteiltem Speicher die Schwierigkeit der Kommunikation. Je komplexer die Aufgaben werden, desto fehleranfälliger sind die Lösungen. Insofern sollte jedes Hilfsmittel genutzt werden die Komplexität zu reduzieren, sei es durch einen softwaretechnischen Ansatz oder durch die Verlagerung von Aufgaben, z.B. der Synchronisation, in die Compilierung.

Im Rahmen dieser Arbeit werden Sprachen untersucht, die parallele Programmierung auf der Basis des objektorientierten Programmierparadigmas unterstützen. Welche Ansatzpunkte die Objektorientierung für die Erweiterung in Richtung Parallelverarbeitung liefert, wird im folgenden kurz beschrieben.

Der objektorientierte Ansatz scheint aufgrund mehrerer Eigenschaften von Vorteil zu sein. Die Definition von Klassen ergibt eine naheliegende Partitionierung von Daten und Kontrollflüssen. Die Gruppierung von Daten in semantische Einheiten erlaubt einerseits die Plazierung und Migration von Objekten aufgrund ihres Typs zu entscheiden. Andererseits müssen Datentypen, die große Datenmengen beschreiben, wie beispielsweise Matrizen, in kleinere, nebenläufig bearbeitbare Verteilungseinheiten zerteilt werden können.

Ein Objekt kann als ein Monitor aufgefaßt werden. Dies ist einerseits eine einfache Methode der Synchronisation. Andererseits wird dadurch die Möglichkeit der Parallelisierung in den Fällen unnötig eingeschränkt, in denen durch die nebenläufige Abarbeitung von Methodenaufrufen die Datenkonsistenz oder die Semantik der Programmausführung nicht gefährdet ist.

Das Konzept des Methodenaufrufs kann leicht auf die Basiskommunikationsprimitiven „send“ und „receive“ abgebildet werden. Die Erweiterung des Kommunikationsmodell um asynchrone Nachrichten fügt sich problemlos ein. Zudem eignet sich dieser Programmieransatz für das Programmieren im großen.

Im folgenden Kapitel 2 werden prinzipielle Unterschiede der Spracherweiterungen untersucht in Hinblick auf ihr Programmiermodell, die Realisierung von Parallelität bzw. Nebenläufigkeit, die Synchronisation bzw. Kommunikation, Zerteilungs- und Verteilungskonstrukte. Danach werden die untersuchten parallelen objektorientierten Sprachen einzeln vorgestellt (siehe Kapitel 3). Abschließend in Kapitel 4 werden diese Sprachen anhand ihrer Unterschiede in Bezug auf die Erweiterungen einander gegenübergestellt. Die Unterschiede in der Realisierung des objektorientierten Paradigmas werden nicht untersucht.

[1] Parallelität: Prozesse werden gleichzeitig auf unterschiedlichen Prozessoren abgearbeitet.
[2] Nebenläufigkeit: Prozesse werden entweder gleichzeitig – wenn dies semantisch möglich ist – auf unterschiedlichen Prozessoren oder beispielsweise im Timesharing-Verfahren auf einem Prozessor abgearbeitet.

2 Taxonomie paralleler objektorientierter Sprachen

Die Integration von Parallelverarbeitung in objektorientierte Sprachen kann auf verschiedene Weisen erfolgen. Diese werden im folgenden beschrieben. Bei der Integration von Parallelität und Objektorientierung lassen sich drei unterschiedliche Herangehensweisen unterscheiden:

- Bereitstellen von Bibliotheksroutinen,
- Erweiterung einer Sprache durch neue Konstrukte und
- Definition einer neuen Sprache.

Die einfachste Möglichkeit ist es, mittels *Bibliotheksroutinen* Parallelarbeit zu ermöglichen, da bei diesem Ansatz kein neuer Compiler nötig ist. Dies ist vor allem im Hinblick auf die Kurzlebigkeit paralleler Architekturen ein wichtiger Gesichtspunkt. Die Erzeugung von parallelen Aktivitäten, die Synchronisation und Kommunikation werden bei der Übersetzung direkt auf Betriebssystemaufrufe abgebildet. Für die Programmierung dieser Aufgaben werden keine weiteren Hilfsmittel zur Verfügung gestellt, d.h. eine Reduzierung der Komplexität findet im allgemeinen nicht statt.

Die *Erweiterung einer objektorientierten Sprache* erfordert die Konstruktion eines neuen Compilers. Dadurch läßt sich aber mehr Einfluß auf den erstellten Code nehmen. Dies kann sich vorteilhaft auf die Effizienz auswirken, z. B. durch compilerinterne Optimierungen. Und dem Compiler können bestimmte Aufgaben, wie die Organisation der Kommunikation, zugewiesen und damit die Komplexität der Programmierung reduziert werden.

In objektorientierten Sprachen liegt es nahe, das Konzept der Vererbung zu nutzen und das zur Parallelverarbeitung nötige Instrumentarium in Basisklassen zu integrieren. Die Klassen von Anwendungsprogrammen erben dann die Eigenschaften und Methoden der Basisklassen. Diese Basisklassen können als Teil des Laufzeitsystems, als Bibliotheksklassen oder als von der Sprache vorgegebene Klassen konzipiert werden. Wird die Bereitstellung solcher Klassen in eine Laufzeitumgebung verlagert, so verlagert sich auch der Aufwand der Integration in die Erstellung der Umgebung. Bei geschickter Konzeption kann der hardwareabhängige Teil so zusammengefaßt werden, daß der Rest portabel ist.

Diese Ansätze haben bei der Verwendung verbreiteter Programmiersprachen den Vorteil, daß große Teile der neu entstandenen Sprache bereits bekannt sind.

Die *Definition einer neuen Sprache* bietet hingegen alle Freiheiten einer bestmöglichen Integration, kann aber auf keinen Vorleistungen aufbauen.

2.1 Programmiermodelle

Modelle für paralleles Programmieren sind unter anderem abhängig von den Modellen paralleler Maschinen. Daher werden diese Modelle hier kurz eingeführt.

Rechnerarchitekturen lassen sich nach der Taxonomie von Flynn äußerst grob in vier Kategorien einteilen (S: Single; M: Multiple; I: Instruction; D: Data):

- SISD: von-Neumann-Rechner
- SIMD: Felder von Prozessoren, Vektorrechner (Pipeline-Rechner)
- MISD: –

- MIMD: Datenflußrechner, Multiprozessoren, massiv-parallele Systeme

Für parallele Architekturen sind nur SIMD- und MIMD-Architekturen relevant. *SIMD-Architekturen* manipulieren in einem Schritt möglichst viele Datenelementen mittels ein und demselben Befehl. Dazu werden die zu manipulierenden Daten vom gemeinsamen, virtuell gemeinsamen oder verteiltem Speicher auf die einzelnen *Rechnerknoten*[3] kopiert. Die als nächstes auszuführende Instruktion wird vom Speicher in eine zentrale Einheit geladen, von dort aus an alle Knoten weitergegeben und dann auf die jeweiligen Daten angewendet. SIMD-Architekturen eignen sich daher besonders für solche Problemstellungen, bei denen auf Elementen großer Datenmengen immer wieder die gleichen Rechenoperationen ausgeführt werden müssen.

Bei der Umsetzung sequentieller Programmiermodelle in ein SIMD-Modell muß der Kontrollfluß vervielfacht und die Datenelemente in soviel *Partitionen*[4] wie Knoten vorhanden aufgeteilt werden. Die Synchronisation ist vom Modell vorgegeben und sehr restriktiv, d.h. Beginn und Ende der einzelnen Instruktionen ist festgelegt. Die Kommunikation kann prinzipiell synchron oder asynchron sein, ist aber im allgemeinen synchron.

MIMD-Architekturen zeichnen sich dadurch aus, daß während eines Zeitintervalls viele Datenelemente anhand vieler verschiedener Instruktionen manipuliert werden. Die einzelnen Kontrollflüsse werden aus einem gemeinsamen, virtuell gemeinsamen oder verteilten Speicher in die Prozessoren des Systems geladen und auf die jeweiligen Daten angewendet.

Diese Architekturform kann grob nach der physikalischen Speicherverteilung eingeteilt werden. Man spricht von *Multiprozessorsystemen*, wenn mehrere Prozessoren auf einen zentralen gemeinsamen Speicher Zugriff haben. Im Gegensatz dazu spricht man von *massiv-parallelen* Systemen, wenn der Speicher physikalisch verteilt ist. *Systeme mit gemeinsamem Speicher* (engl.: shared memory systems) haben den Vorteil eines globalen Adreßraums und damit der Verwendbarkeit herkömmlicher Programmiersprachen bzw. -stile, z.B. Fortran, oder C++. Der Nachteil ist eine geringe Skalierbarkeit. Die maximale Prozessorzahl liegt derzeit bei 30 Prozessoren [Giloi 93].

Bei *Systemen mit verteiltem Speicher* (engl.: distributed memory systems) ist die Skalierbarkeit ein wesentlicher Vorteil gegenüber Systemen mit gemeinsamen Speicher. Es ist fast jede beliebige Systemgröße denkbar, da bei diesen Systemen das Verhältnis von Verarbeitungsbandbreite des Prozessors zur Zugriffsbandbreite des Knotenspeichers konstant ist. Der Nachteil ist, daß herkömmliche Programmiermodelle gar nicht oder nicht ohne Erweiterungen verwendet werden können. Zudem ist die Partitionierung der Anwendungsprogramme, sowohl im Hinblick auf die Daten als auch die Kontrollflüsse ein nur schwierig zu automatisierender Vorgang, da NP-vollständig. Die jeweiligen Compiler müssen sich auf mehr oder weniger gute Heuristiken stützen. Programmierer oder Compiler müssen die Partitionierung des Anwenderprogramms in eine Vielzahl kooperierender Prozesse durchführen, wobei die Programmierung explizit die Kommunikation zwischen diesen Prozessen und damit deren Synchronisation einschließen muß. Daraus ergibt sich die Notwendigkeit der Erweiterung herkömmlicher Programmiersprachen. Daß mit der Kommunikation auch zumeist eine Synchronisationsaktion verbunden ist, ist einer der Kritikpunkte an den bisherigen Programmiermodellen für parallele Systeme [Giloi 93].

---

[3] Ein Rechnerknoten, kurz Knoten, umfaßt mindestens einen Prozessor und lokalen Speicher.

[4] Eine Partition besteht aus einem Teil der Daten und den dazugehörigen Methoden des Gesamtprogramms. Alle Partitionen zusammen bilden das Gesamtprogramm. Eine Partitionierung der Daten und ihrer Kontrollflüsse soll zu einer möglichst hohen Leistungssteigerung führen.

Anwendungsprobleme, die sich für eine Parallelisierung eignen, zeichnen sich zum einen durch zumeist große, homogene oder heterogene Datenmengen aus, die untereinander ein gewisses Maß an Unabhängigkeit aufweisen, so daß einzelne Datenkomponenten gleichzeitig manipuliert werden können. Homogene Datenmengen finden sich beispielsweise bei Vektoren, Matrizen oder partiellen Differentialgleichungen. Koordiniert werden hier bestimmte aufeinanderfolgende Berechnungsabschnitte. Heterogenene Daten finden sich beispielsweise bei der Steuerung bzw. Regelung von technischen Prozessen, wobei das Eintreffen bestimmter Ereignisse, z.B. das Überschreiten eines bestimmten Zeitrahmens, zur Koordination von Aktionen führt.

Das Maß der Abhängigkeit bzw. Unabhängigkeiten der Daten und Aktionen voneinander beeinflußt die Entscheidung, welche Eigenschaften eines Programmiermodells und letztendlich welches Maschinenmodell zu einer Umsetzung des Problems nötig sind.

Das SIMD-Modell geht davon aus, daß auf vielen Datenpartitionen von homogenen Daten ein und die selbe Instruktion synchron ausgeführt wird. Die Synchronisierung erfolgt implizit durch die Abarbeitung einer Instruktion nach der anderen. Aufgrund der extremen Feinkönigkeit kann dieses Modell für die Lösung von datenparalleler Problemen mit Hilfe der Objektorientierung im allgemeinen nicht angewendet werden.

Statt nur eine Instruktion synchron auszuführen, wird eine Folge von Instruktion, wie sie z.B. ein Methodenaufruf umfassen könnte, auf vielen gleichstrukturierten Datenpartitionen asynchron zueinander ausgeführt. Diese führt zu einem erweiterten SIMD-Modell, dem sogenannten SPMD-Modell[5]. Die zusammengefaßten Instuktionsfolgen bestehen nach [Giloi 93] aus drei Phasen: einem definierten gemeinsamen Beginn, einem asynchronen Abarbeiten der Instruktionsfolge und einer Abschlußsynchronisation, um die Beendigung aller Instruktionsfolgen ab einem bestimmten Zeitpunkt, d.h. vor der Fortsetzung nachfolgender Aktionen, sicherzustellen.

MIMD erweitert das SIMD-Modell insofern, als unterschiedliche Instruktionen auf unterschiedliche Daten während eines Zeitintervalls ausgeführt werden, das etliche Maschinenbefehlszyklen umfassen kann. Dabei werden die einzelnen Instruktionen nicht durch den Takt eines Maschinenbefehlszyklus synchronisiert, sondern werden asynchron zueinander abgearbeitet. Wie für SIMD gilt auch für das MIMD-Modell, daß im Rahmen der Objektorientierung Instruktionen eine zu feinkörnige Struktur darstellen. Aufgrund der asynchronen Abarbeitung von einzelnen Kontrollflüssen ergibt sich durch die Bündelung von Instruktionen kein prinzipiell anderes Modell. Wichtig ist, daß durch das Modell weder Beginn noch Ende unterschiedlicher Aktionen in Beziehung gesetzt werden. Für datenparallele Anwendungen ist dieses Modell schlechter geeignet als das SPMD-Modell. Andererseits ist eine Umsetzung von Programmen für MIMD-Architekturen auf Arbeitsplatzrechnernetze einfacher als für SPMD-Programme.

Parallelrechnern ist häufig ein Vorrechner vorgeschaltet, von dem aus das Programm auf den Parellelrechner geladen wird. Eines der offenen Probleme ist die Organisation der Initialisierung, d.h. welche Daten auf welche Knoten verteilt werden, um eine möglichst effiziente Bearbeitung zu ermöglichen. Genauso ungeklärt ist die Verteilung von Programmsequenzen während der Programmausführung.

[5] SingleProgramMultipleData-Programmiermodell

2.2 Nebenläufigkeitsmodelle

Um eine Steigerung der Effizienz durch nebenläufige Ausführung zu erreichen, müssen sich Teile eines Anwendungsprogramms parallel ausführen lassen. Damit etwas parallel stattfinden kann, muß ein Grad an Unabhängigkeit zwischen den einzelnen Teilen vorhanden sein, d.h. die nötigen Betriebsmittel müssen vorhanden sein. Betriebsmittel sind hier neben den Prozessoren und dem benötigten Speicher vor allen die aufzuteilenden Daten, z.B. Zwischenergebnisse oder Zustände. (Wie Daten aufgeteilt werden können, wird in den nächsten Abschnitten erläutert.)

Ein wichtiges Maß für die erreichbare Parallelität ist die Anzahl von unabhängig zueinander ausführbaren *Aktivitäten*. Durch nebenläufige Abarbeitung von Aktivitäten wird die erreichbare Parallelität beschränkt. Nach den Teilen eines Programms, die sich zu Aktivitäten bündeln lassen, lassen sich folgenden Aktivitätsebenen unterscheiden[6]:

- Prozesse,
- Anweisungen und
- Elementaroperationen.

Der Anzahl der Aktivitäten, die gleichzeitig ausgeführt werden können, steht der Aufwand für die Erzeugung und Verwaltung von Prozessen, deren Kommunikation und Synchronisation entgegen. Eine Effizienzsteigerung läßt sich nur erzielen, wenn der Verwaltungsaufwand die Leistungssteigerung durch die Parallelisierung nicht wieder zunichte macht. Folgende *Granularitätsstufen* lassen sich unterscheiden:

- grobkörnig, d.h. die einzelnen Prozesse eines Anwendungsprogramms,
- feinkörnig, d.h. die Anweisungen eines Basisblocks[7] oder einer Schleife oder eines leichtgewichtigen Prozesses[8], und
- sehr feinkörnig, d.h. die Teiloperationen eines zusammengesetzten Ausdrucks oder einfache Anweisungen.

Ein Ausführungsmodell kann passive oder aktive Objekte vorsehen. Innerhalb des Modells der passiven Objekte werden von den Klassen unabhängige Prozesse erzeugt, die Objektdaten nach ihren Methoden manipulieren. Innerhalb des Modells der aktiven Objekte werden Objekte als Prozesse realisiert, die nach ihrer Erzeugung aktiv werden.

Die Granularität des Ausführungsmodells mit passiven Objekten ist einerseits abhängig von der Anzahl der Prozesse und andererseits davon, wieviele Prozesse gleichzeitig auf ein Objekt zugreifen können. Die Granularität des Ausführungsmodells mit aktiven Objekten führt im allgemeinen zuerst zu einer grobkörnigen Granularität, da ein oder mehrere Objekte auf einen Prozeß abgebildet werden. Diese grobkörnige Granularität kann durch die nebenläufige Bearbeitung von feinkörnigeren Strukturen gemildert werden.

Nach [Booch 91] sollen innerhalb einer gegebenen Klassen alle Operationen/Methoden primitiv gehalten werden, so daß jede ein wenig umfangreiches und damit wohldefiniertes Verhalten zeigt. Solche Methoden werden als feinkörnig bezeichnet. Insofern kann der Aufruf einer Methode entweder durch einen nebenläufigen Prozeß manipuliert oder durch die Erzeugung eines leichtgewichtigen Prozesses realisiert werden.

---

[6] Parallelarbeit läßt sich auch durch parallele Ausführung ganzer Anwendungsprogramme organisieren. Dieser Fall wird hier nicht betrachtet.

[7] Ein Basisblock ist die Anweisungsfolge zwischen zwei Verzweigungsbefehlen. Ein Basisblock unterscheidet sich von einem leichtgewichtigen Prozeß insofern, als daß hier keine Sprünge vorkommen dürfen.

[8] Leichtgewichtige Prozesse sind solche, die sich mit anderen leichtgewichtigen Prozessen eine Adreßbereich teilen.

Eine feinerkörnige Granularität ist innerhalb des Modells der aktiven Objekte aufgrund des zu erwartenden hohen Verwaltungsaufwands nicht sinnvoll, außer die Zielarchitektur bietet gerade dafür effiziente (Hardware-) Unterstützung. Als Programmiermodell kann ein solches Vorgehen durchaus sinnvoll sein, wenn z.B. der Compiler die einzelnen Objekte zu größeren Einheiten bündelt.

Nach [Giloi 93] ergibt sich folgender Zusammenhang zwischen Granularität und Parallelität:

Programmebene	Kennzeichnung	Parallelität
Prozesse	mehrere Prozesse laufen gleichzeitig ab	hoch
Anweisungen	mehrere Anweisungen eines Programms werden gleichzeitig ausgeführt	niedrig: Basisblöcke hoch: Schleifen hoch: leichtgewichtige Prozesse
Elementaroperationen	mehrere Operationen eines zusammengesetzten Ausdrucks werden gleichzeitig ausgeführt	niedrig

Tabelle 1: Zusammenhang zwischen Granularität und Parallelisierung

Egal auf welcher Ebene Aktivitäten betrachtet werden, ergibt sich aus der Notwendigkeit, Informationen auszutauschen, der Zwang, die Parallelität einzuschränken. Die sich ergebenden Probleme der Synchronisation und Kommunikation werden in den beiden anschließenden Abschnitten behandelt.

2.2.1 Synchronisation

Synchronisation dient zum einen zur Erhaltung der Datenkonsistenz und zum anderen zur Einhaltung eines semantisch korrekten Kontrollflusses. Dabei verschmilzt Synchronisation in Systemen mit verteiltem Speicher im allgemeinen mit Kommunikationsmechanismus (z.B. Rendezvous). Dieses Phänomen wird im nächsten Abschnitt behandelt.

Zur Erhaltung der Datenkonsistenz werden die Zugriffe sequentialisiert, üblicherweise mit Semaphoren, Monitoren oder ähnlichem. Diese Mechanismen sequentialisieren den Zugriff auf Daten, indem immer nur ein Prozeß in den kritischen Abschnitt eintreten darf. Weitere Prozesse werden blockiert und in eine Warteschlange eingereiht. Einfache Objekte können sich wie Monitore verhalten, indem sie lokale wie entfernte Methodenaufrufe sequentiell in der Reihenfolge ihres Eintreffens abarbeiten. Zur Flexibilisierung des Konzepts und damit zur Erhöhung der Parallelität kann unterschieden werden zwischen:

- Zugriffsarten: lesend oder schreibend, d.h. nur schreibende Prozesse führen zu Sequentialisierung,
- Zugriffszeitpunkt: Initialisierung von einmal beschreibbaren Variablen bei Programmstart oder verzögert nach Bereitstellung eines Zwischenergebnisses, d.h. eine nichtinstanziierte Variable führt zur Blockierung lesender Prozesse, und
- Zugriffshäufigkeit: einmal oder mehrmals beschreibbar, einmal oder mehrmals lesbar.

Bei nur lesbaren und beim Programmstart initialisierten Daten müssen parallele bzw. nebenläufige Zugriffe nicht synchronisiert werden. Variablen mit verzögerter Initialisierung (engl.:

future variable) blokkieren Prozesse solange, bis sie einen gültigen Wert haben. Nach diesem Zeitpunkt bedarf es keiner weiteren Synchronisation. Daten, die zwar mehrmals verändert werden und nur einmal gelesen werden können, benötigen die üblichen Mechanismen zur Synchronisation der Schreibzugriffe und einen zum Ungültigmachen nach dem einmaligen Lesezugriff. Desweiteren kann bei veränderbaren Daten der Schreibzugriff auf den Prozeß beschränkt werden, dem diese Daten zugeordnet sind bzw. in dessen Speicherbereich diese Daten liegen (engl.: owner computation). Wollen andere Prozesse schreibend auf solche Daten zugreifen, müssen diese in den jeweiligen Speicherbereich verbracht werden.

Bei Systemen mit verteiltem Speicher werden diese Konzepte meist in die entsprechenden Aktualisierungsstrategien der Caches abgebildet, wobei sich auch weniger restriktive Methoden in Bezug auf die Datenkonsistenz implementieren lassen (siehe z. B. [Giloi 93]).

Für die Leistungssteigerung mittels paralleler Ausführung von Kontrollflüssen ist es am günstigsten, wenn der Start- und Terminationszeitpunkt von Aktivitäten beliebig ist und nur die Abwicklungsgegebenheiten der Kommunikation die Parallelität eingeschränkt. Desweiteren wird die Parallelität eingeschränkt, wenn es einen definierten Start- oder Terminationszeitpunkt für mehrere oder alle Aktivitäten gibt. Hier kann man von asynchronen Prozessen bzw. asynchronen leichtgewichtigen Prozessen sprechen. Noch weiter geht die Einschränkung, wenn es sowohl einen festen Start- als auch einen festen Beendigungszeitpunkt gibt. Diese Aktivitäten lassen sich als synchrone Prozesse bzw. synchrone leichtgewichtige Prozesse bezeichnen. Zur Synchronisation des Kontrollflusses lassen sich folgende Unterscheidungen treffen:

- festgelegter Beginn, d.h. vor einem bestimmten Zeitpunkt innerhalb der Porgrammausführung wird keine dieser Aktivitäten gestartet,
- festgelegtes Ende, d.h. bevor nicht alle Aktivitäten beendet wurden, wird in der Programmausführung nicht mit anderen Aktivitäten fortgefahren, und
- festgelegter Beginn und Ende paralleler Aktivitäten.

Beispiele für letztgenannte Möglichkeiten sind nebenläufige Blöcke von Aktivitäten, die mit „cobegin“ und „coend“ begrenzt werden, oder Schleifen, deren einzelne Schleifendurchläufe parallel zueinander abgearbeitet werden und deren Bearbeitung erst terminiert, wenn alle einzelnen Schleifendurchgänge abgearbeitet wurden. Berechnungen mit hoher Datenparallelität fallen im allgemeinen in diese Kategorie. Die Parallelität von Aktivitäten innerhalb eines Blocks mit festgelegtem Beginn und Ende kann durch Datenabhängigkeiten eingeschränkt sein.

Aktivitäten mit festgelegtem Ende tragen meist ein Teilergebnis zu einem Gesamtergebnis bei, wobei der Berechnungsbeginn unwesentlich ist, d.h. es bedarf keiner gemeinsamen Initialisierung. Aktivitäten mit festgelegtem Beginn berechnen ein Ergebnis, das aber für den weiteren Verlauf der diese auslösenden Aktivität nicht benötigt wird, z.B. das Aktualisieren replizierter Daten.

2.2.2 Kommunikation

In Systemen mit verteiltem Speicher, deren Laufzeitumgebung auch keinen virtuell gemeinsamen Speicher zur Verfügung stellt, muß der Datenaustausch explizit auf der Sprachebene organisiert werden. Mit dem Austausch von Daten verbinden sich Synchronisationseigenschaften, da zum Zustandekommen einer Kommunikation der empfangende Prozeß an einem Punkt angelangt sein muß, an dem er dazu in der Lage ist. Die Daten müssen vorhanden, also Berechnungen abgeschlossen sein, bevor diese versendet werden können.

Die Notwendigkeit zu kommunizieren schränkt die Parallelität ein. Insofern müssen die Kommunikationsoperationen so beschaffen sein, daß die Synchronisierung und damit die Einschränkung der Nebenläufigkeit auf ein notwendiges Minimum reduziert wird. Im folgenden werden verschiedene Kommunikationsarten vorgestellt.

Für einen Datenaustausch genügt es, wenn der Sender eine Nachricht, d.h. die auszutauschenden Daten, an den Empfänger weiterleitet. Benötigt der Sender vom Empfänger eine Antwort, so sendet dieser nach einer bestimmten Zeit eine Nachricht an jenen. Eine solche Kommunikation besteht auf Senderseite aus einer Sende- und Empfangsoperation und auf Empfängerseite aus einer Empfangs- und Sendeoperation. Das Senden und Empfangen läßt sich jeweils blockierend und nichtblockierend durchführen.

Man spricht von blockierendem Senden, wenn der Sender solange blockiert ist bis der Empfänger empfangesbereit ist. Beim blockierenden Empfangen wird ein Empfänger solange blokkiert bis eine Nachricht vom Sender eintrifft. Beim nichtblockierenden Senden und Empfangen kann ohne die entsprechende Nachricht weitergearbeitet werden. Mit Hilfe der Sende- und Empfangsprimitiven können folgende Kommunikationsarten realisiert werden:

- synchron = blockierendes Senden und blockierendes Empfangen: Ein Sender einer Nachricht ist solange blockiert, bis der Empfänger bereit ist, die Nachricht entgegenzunehmen. Der Empfänger ist solange blockiert, bis der Sender bereit ist, eine Nachricht zu versenden. Für das Versenden einer Antwort gilt entsprechendes.
- asynchron (Variante 1) = zeitweilig nichtblockierendes Senden und blockierendes Empfangen: Ein Sender verschickt seine Nachricht und setzt seine Berechnungen solange fort, bis die Antwort vom Empfänger benötigt wird. Diese Kommunikationsart ist gekoppelt mit einer entsprechenden Zugriffssynchroniation für die Variablen (future variables), die die Antworten zu einem späteren Zeitpunkt enthalten werden.
- asynchron (Variante 2) = nichtblockierendes Senden und blockierendes Empfangen: Ein Sender verschickt seine Nachricht und setzt seine Berechnungen fort. Auf eine Antwort wird auch zu einem späteren Zeitpunkt nicht gewartet. Der Empfänger wartet auf eine Nachricht und führt nach dem Empfangen entsprechende Berechnungen aus. Die Ergebnisse dieser Berechnungen können beispielsweise an Dritte weitergeschickt werden.

Bei diesen Kommunikationsarten spielt es keine Rolle, ob der Empfänger eine Empfangsroutine aufruft oder durch das Eintreffen einer Nachricht vom Betriebsystem zu einem bestimmten Zeitpunkt in Kenntnis gesetzt wird.

Wird die Kommunikation explizit programmiert, müssen Probleme wie beispielsweise Verklemmungen, ungültige Referenzen auf bereits terminierte Prozesse usw. auch explizit behandelt werden. Der Methodenaufruf der objektorientierten Programmierung läßt sich verhältnismäßig einfach auf das Senden einer Nachricht abbilden.

2.3 Zerteilung und Verteilung

Paralleles Arbeiten setzt zum einen die *Zerteilung*[9] des Kontrollflusses und/oder auch der Daten und zum anderen die *Verteilung*[10] der einzelnen Kontrollfluß[11]- und Datenpratitionen auf die zur Verfügung stehenden Knoten voraus. Beim Entwurf eines Algorithmus kann davon ausgegangen werden, daß es virtuell genau so viele Knoten wie parallel ausführbare Aktivitäten gibt. Im Normalfall wird die maximale Anzahl der Aktivitäten die der vorhandenen Knoten eines Systems bei weitem übersteigen, d.h. ein Teil Aktivitäten, die eigentlich gleichzeitig stattfinden könnten, können nur quasiparallel abgearbeitet werden.

In diesem Abschnitt werden die Zerteilungs- und Verteilungskonstrukte auf Sprachebenen betrachtet. Die Verteilung von zu erzeugenden Prozessen zur Laufzeit nach Gesichtspunkten der Laufzeit und Prozeßmigration werden nicht betrachtet.

Innerhalb des objektorientierten Ansatzes werden aktive und passive Objekte durch Prozesse realisiert bzw. manipuliert. Folgende Abbildungsmöglichkeiten von Objekten auf Prozesse sind möglich (n: Anzahl der leichtgewichtigen Prozesse; N: Anzahl der Prozesse; i: Anzahl der Objekte):

1: 1 ein Objekt wird als ein eigenständiger Prozeß ausgeführt,
1: n ein Objekt wird durch einen Prozeß ausgeführt, der sich wiederum aus n nebenläufigen, leichtgewichtigen Prozessen zusammensetzt,
1: N ein Objekt wird durch N Prozesse ausgeführt,
i: 1 i Objekte werden durch einen Prozeß ausgeführt und

Im folgenden werden zuerst aktive und anschließend passive Objekte betrachtet. Die einfachste Möglichkeit ist die Realisierung eines aktiven Objekts durch einen Prozeß (1:1). Die Erzeugung von Prozessen ist im allgemeinen mit einem gewissen Aufwand verbunden. Dieser Aufwand sollte im Verhältnis stehen zum Effizienzgewinn der mit der Prozeßerzeugung verbundenen Parallelisierung und rentiert sich somit nur für Objekte ab einem gewissen Umfang.

Objekte kapseln Daten. Das erschwert den Zugriff auf diese Daten durch Methoden anderer Objekte. Daher kann es günstig sein, an Methoden umfangreiche Objekte zu spezifizieren und den Verlust an Parallelität durch nebenläufige leichtgewichtige Prozesse auszugleichen (1:n). Eine Methode kann dann als ein leichtgewichtiger Prozeß realisiert werden. Steht eine Ressource, z.B. eine Nachricht, nicht zur Verfügung, blockiert nicht das ganze Objekt, sondern nur ein leichtgewichtiger Prozeß. Das Umschalten zwischen leichtgewichtigen Prozessen ist aufgrund des geminsamen Prozeßbeschreibungsblockes einfach und effizient zu realiseiren. Der Zugriff auf die Daten ist aufgrund des gemeinsamen Adressraums lokal. Damit vereinfacht sich die Kommunikation erheblich. Andererseits ist je nach Art der Zugriffe wiederum Synchronisation nötig.

Bei Objekten, die sehr große Datenmengen umfassen, z.B. Vektoren oder Matrizen, werden diese Daten aufgeteilt und der Kontrollfluß vervielfältigt (1:N). Eine Datenpartition mit einem duplizierten Kontrollfaden wird dann durch einen Prozeß realisiert. Im allgemeinem ergeben sich so viele Prozesse wie Knoten im System vorhanden sind. Die Datenpartitionen sind im

---

[9] Zerteilung (oder Partitionierung) beschreibt die Art und Weise wie der Kontrollfluß in einzelne Aktivitäten zerteilt wird bzw. wie zusammengehörige Daten in mehrere Partitionen zerteilt werden, so daß sie nebenläufig manipuliert werden können.

[10] Verteilung beschreibt die Art und Weise wie Kontrollfluß- und Datenpartitionen auf die zur Verfügung stehenden Knoten zur Bearbeitung verteilt werden.

[11] Aktivitäten sind Partitionen des Gesamtkontrollflusses.

allgemeinen so unabhängig voneinander, daß der verteilte Adreßraum sich nicht nachteilig auswirkt.

Die Bündelung von Objekten zu einem Prozeß kann sinnvoll sein, wenn das Programmiermodell von sehr feinen Objekten ausgeht, so daß eine 1:1-Umsetzung außer durch spezielle Hardwareunterstützung nicht effizient realisiert werden kann. Oder wenn zur Vereinfachung der Kommunikation, z.B. der Unterscheidung zwischen lokalen und entfernten Methodenaufrufen, mehrere Objekte auf einen Prozeß abgebildet werden (N:1). Ein Prozeß kann immer nur von einem Knoten zu einer Zeit abgearbeitet werden.

Ein passives Objekt kann von einem (1:1) oder von mehreren Prozessen (1:N) bzw. mehreren leichtgewichtigen Prozessen (1:) gleichzeitig manipuliert werden. Dies ist abhängig von der Anzahl der verfügbaren Prozesse und von der Datenabhängigkeit innerhalb des Objekts. Bei Objekten mit sehr umfangreichen Datenstrukturen können wiederum mehrere Prozesse jeweils eine Datenpartition bearbeiten (1:N).

Die Realisierung von mehreren aktiven wie passiven Objekten durch einen Prozeß (i:1) kommt einer Sequentialisierung gleich und ist aus semantischen Gründen für bestimmte Programmabschnitte nötig.

Für die optimale Ausnutzung vorhandener Ressourcen ist es wichtig, Partitionen von Daten und Kontrollflüssen so zu verteilen, daß möglichst viele Berechnungen gleichzeitig stattfinden können und möglichst wenig kommuniziert werden muß. Die Partitionierung von Anwendungsprogrammen ist oft von deren Semantik abhängig und wird bisher in Ermangelung an automatisierenden Mechanismen von Hand kodiert. Da aber die meisten Parallelrechner über sehr viele Knoten verfügen, die mit den unterschiedlichsten Kommunikationsmedien verbunden sind, sollte andererseits möglichst viel Verteilungsarbeit automatisiert werden. Zwischen den Extremen völliger Verteilungstransparenz und der völligen Aufgabe dieser können unterschiedliche Verteilungskonzepte angesiedelt werden:

- Plazieren von einem Objekt[12] auf einem bestimmten Knoten,
- Verteilung von Objekten einer Klassen mit bestimmten Eigenschaften,
  - Verteilung von vervielfachten Objekten auf alle oder möglichst vielen Knoten[13],
  - Verteilung von Objekten auf Knotencluster nach ihrem Kommunikationsverhalten,
  - Verteilung von Partitionen auf Knoten nach deren Entfernungsrelationen,
- Verteilung von Objekten zur Laufzeit nach Strategien der Lastverteilung.

Durch explizite Nennung einer Knotenkennung kann eine Partition auf genau diesem Knoten plaziert werden. Dies kann sich für heterogene Architekturen, d.h. solche mit Spezialprozessoren, oder für kleine Architekturen mit überschaubarer Prozessoranzahl als nützlich herausstellen. Allerdings haben nur Architekturen mit gemeinsamem Speicher eine verhältnismäßig überschaubare Prozessorzahl von maximal 30. Für homogene Architekturen mit vielen tausenden Knoten ist dieses Vorgehen als einziges Verfahren wegen seiner Unübersichtlichkeit nicht praktikabel.

Prozesse lassen sich nach ihren Eigenschaften auf Knoten verteilen. Prozesse, die als Dienstabwickler (engl.: server) genutzt werden können und keinen eigenen Zustand haben, können zur Vermeidung von entfernten Methodenaufrufen auf allen oder auf den Knoten plaziert werden, auf dem sie voraussichtlich benötigt werden. Ebenso können gerade Prozesse, die Teilergebnisse liefern, überall dort plaziert werden, wo diese benötigt werden. Die Teilergeb-

[12] Objekte stellen für sich schon eine Partitionierung von Daten und des Kontrollflusses dar.
[13] Innerhalb von fehlertoleranten Anwendungen werden replizierte Objekte auf wenige Knoten verteilt.

nisse werden mit einer Kommunikationsanweisung zu einem Gesamtergebnis zusammengefügt.

In der Regel werden nicht alle Bearbeitungsschritte, die parallel ausgeführt werden könnten, auch parallel ausgeführt, d.h. sie werden zwangssequentialisiert. Für die Verteilung könnten solche Prozesse auf einen Knoten gelagert werden, die nie gleichzeitig aktiv sein können, z.B. solche, deren Kommunikation untereinander synchron abgewickelt wird. Bei Sprachen mit nur synchroner Kommunikation hilft das nicht weiter. Dort müßte man feststellen können, wie häufig Objekte miteinander Daten austauschen. Je nach Kommunikationshäufigkeit könnten Objekte auf Knoten plaziert werden, die physikalisch entsprechend dicht zueinander liegen.

Wird das Kommunikationsverhalten in Betracht gezogen, können Prozesse je nach Kommunikationsaufkommen auf den selben, auf benachbarte oder auf weit entfernte Knoten plaziert werden. Ein weiterer Schritt wäre die Betrachtung der Kommunikationshardwarestruktur der Zielarchitektur. Dies würde eine noch weitergehende Aufgabe der Verteilungstransparenz bedeuten. Voraussetzung ist die Möglichkeit, das Kommunikationsverhalten statisch abzuschätzen.

Die Frage ist, was man mit der unterschiedlichen Anzahl von Prozessen auf den jeweiligen Knoten macht, ob man diese mit Hilfe von Lastverteilungsstrategien aneinander angleicht oder ob man die Verteilungskriterien weiter verfeinert. Da weder das gesamte Verhalten eines Prozesses statisch abschätzbar ist, noch bei der Möglichkeit der dynamischen Prozeßerzeugung die Anzahl der Prozesse vorherbestimmbar ist, werden Lastverteilungsverfahren benützt, um dynamisch Prozesse auf Knoten zu verteilen.

3 Beschreibung paralleler objektorientierter Sprachen

Im folgenden werden mehrere objektorientierte parallele Sprachen vorgestellt. Die Reihenfolge richtet sich nach der Herangehensweise der Spracherweiterung, wie sie zu Beginn von Kapitel□2 beschrieben wurde. Innerhalb der Abschnitte werden die Sprachen lexikalisch geordnet.

3.1 Spracherweiterung durch Bereitstellen von Bibliotheksroutinen

3.1.1 EPEE

EPEE (Eiffel Parallel Execution Environment) basiert auf Eiffel, könnte aber auch auf jeder anderen objektorientierten Sprache basieren, die folgende Eigenschaften hat: strenge Datenkapselung einhergehend mit statischer Typanalyse, Mehrfachvererbung, dynamisches Binden und Generizität.

Ziel von EPEE ist es, Programme so zu implementieren, daß sie bei linearem Effizienzgewinn auf einer beliebig skalierbaren Anzahl Prozessoren mit verteiltem Speicher abgearbeitet werden können [Jézéquel 93]. Dabei soll mit Hilfe bereitgestellter Basisklassen eine vollständige Verteilungstransparenz erreicht werden.

Dem Paradigma der Funktionsparallelisierung (MIMD) wird das der Datenparallelisierung (SPMD) vorgezogen. Ein Objekt mit vielen regulären Daten wird durch viele Prozesse realisiert.

Aus der Verteilung der Daten auf alle Knoten, die dann alle die gleiche Programmsequenz abarbeiten, ergibt sich – zumindest theoretisch – die Skalierbarkeit der Knotenanzahl. Dabei soll die Partitionierung (Zerteilung) der Daten weder manuell noch vom Compiler vorgenommen werden. Dies soll vielmehr in einer Basisklasse verborgen werden, die allgemeine Verteilungsstrategien für Datenstukturen enthält.

Die Klasse „Distributed_Aggregate“[14] stellt alle diejenigen für eine transparente Verteilung notwendigen Funktionalitäten bereit. Eine sequentielle Klasse wird mit Hilfe der Vererbung von Eigenschaften der verteilten Klasse und der Klasse „Distributed_Aggregate“ parallelisiert, ohne daß sich ihre Semantik auf Anwendungsebene ändert.

Bei der Klasse „Distributed_Aggregate“ handelt es sich um Aggregate generischer Daten, verteilt auf die jeweiligen Prozessoren. Hinzu kommt eine Menge von Methoden, die es ermöglichen, transparent auf Daten zuzugreifen, sie umzuverteilen, eine Methode auf allen Partitionen auszuführen und assoziative Funktionen auf den Aggregaten zu berechnen. Die Implementierung der Methoden hängt von der zugrundegelegten Architektur ab.

[14] Distributed Aggregates sind beeinflußt von den Ideen der Concurrent Aggregates [Chien 90]

Die Semantik des Datenzugriffs basiert auf dem Refresh/Exec-Konzept [André 90]. Hierbei werden Daten vor einem Lesezugriff mit Hilfe einer Broadcast-Nachricht aktualisiert (Refresh). Schreibend zugreifen (Exec) kann nur der Prozessor, dem die Daten zugeordnet sind. Die Synchronisation des Kontrollflusses geschieht anhand der Gegebenheiten des unterstützten SPMD-Programmiermodells, d.h. Synchronisation von Beginn und Ende parallel ausgeführter Aktivitäten.

Vorteile sind die Transparenz und damit die Einfachheit der Programmierung auf Anwendungsebene, d.h. auf Seiten des Auftraggebers (engl.: client). Auf seiten des Auftragsnehmer (engl.: server) muß dagegen die gesamte Komplexität der Parallelisierung behandelt werden, z.B. der Verteilung auf Knoten ohne gemeinsamen Speicher mit expliziter Kommunikation. EPEE eignet sich vor allem für Probleme mit großen Datenmengen. Das Konzept führt nicht zu einer Spracherweiterung, d.h. neue Compiler sind nicht notwendig. Somit sind solche Programme auch auf einer sequentiellen Maschinen lauffähig durch Weglassen der Verteilungsinformation.

Nachteile sind, daß nur regelmäßige Datenstrukturen in Betracht gezogen werden. Die Schwierigkeiten der parallelen Programmierung werden lediglich verlagert in die verteilten Klassen. Desweiteren wird nicht betrachtet, wann die Kommunikationskosten den Effizienzgewinn durch Parallelisierung überwiegt. Dagegen könnten die Datenverteilung und damit auch die der Kommunikation für die spezifische Architekturen optimiert werden.

3.2 Erweiterungen einer Sprache durch neue Konstrukte

3.2.1 CC++

CC++ [Sivilotti 94a] erweitert die objektorientierte Sprache C++ um generische Bibliotheksfunktionen für paralleles Programmieren. Die Objektorientierung der Basissprache C++ trägt durch generische Klassen und Vererbung maßgeblich zur Integration und Unterstützung der parallelen Komponenten bei.

Als Programmiermodell wird vor allem MIMD, aber auch SPMD, unterstützt. In Bezug auf die Kommunikation werden mehrere Paradigmen bereitgestellt. Damit kann zur Lösung unterschiedlicher Probleme die Programmiersprache beibehalten werden [Sivilotti 94b]. Die Prozeßgranularität kann innerhalb von parallel ausgeführten Blöcken die einer Anweisung oder die eines Methodenaufrufs sein.

Bei den Erweiterungen handelt es sich vor allem um Konstrukte zur parallelen Ausführung von Anweisungsblöcken oder for-Schleifen.

- par: Anweisungen par-Blocks werden parallel zueinander ausgeführt, soweit bestehende Datenabhängigkeiten dies zulassen. Der Block terminiert, sobald alle Anweisungen beendet sind. „Par“ beinhaltet somit eine implizite Synchronisation.
- parfor: Die Durchläufe einer parfor-Schleife werden parallel zueinander ausgeführt. Die Schleife terminiert, wenn alle Schleifendurchläufe beendet sind.
- spawn[15]: Wird eine spawn-Funktion aufgerufen, wird ein vollständig unabhängiger Kontrollfaden erzeugt. Um ein vollständig asynchrones Arbeiten des Eltern-Kontrollflusses zum spawn-Kontrollfluß zu ermöglichen, liefern spawn-Funktionen kein Ergebnis.

[15] wörtlich: laichen, transitiv verwendet im Sinne von: initiieren, in die Welt setzen;

Daneben werden atomare Funktionen und Variablen, die nur ein einziges Mal geschrieben werden können (engl.: single-assignment variables), zur Sicherung der Datenkonsistenz bereitgestellt. Die wichtigsten dies bezüglichen Erweiterungen von CC++ sind (weitere Einzelheiten siehe [Carlin 92]):

- atomic: Eine atomare Funktion kann weder unterbrochen noch vorzeitig beendet werden.
- sync: bezeichnet ein einmal beschreibbares Objekt; Vor allem innerhalb von par-Blöcken und bei spawn-Funktionen werden asynchrone Kontrollflüsse anhand von sync-Variablen zu einem möglichst späten Zeitpunkt synchronisiert.

Für Verteilungseinheiten wird ein Objekttyp „processor“ eingeführt. Ein Prozessorobjekt (engl.: processor object) ist eine Menge von Daten und Berechnungen, welche einen virtuellen Adreßraum definieren. Welche Objekte zu virtuell einem Adreßraum gehören, wird durch die Compilerdirektive „-ptype=“ festgelegt. Über globale Zeiger haben andere Objekte Zugriff auf Methoden und Daten. Jeder der Zugriff auf ein Prozeßobjekt hat, kann mittels synchronem Methodenaufruf dessen Daten manipulieren. Zugriffskonflikte müssen mit Hilfe von „atomic“ und „sync“ ausgeschlossen werden. Auf einem Knoten können sich mehrere Prozessorobjekte befinden.

Eine Klasse wird zu einem Prozessorobjekt, indem der Deklaration das Schlüsselwort „global“ vorangestellt wird. Das Prozessorobjekt definiert die Schnittstelle der Objekte dieser Klasse. Prozessorobjekttypen können vererbt werden.

Erreicht wird durch dieses Konzept die Trennung der Problempartitionierung von der Abbildung der Partitionen auf die vorhandenen Knoten. Die Abbildung erfolgt durch explizite Spezifikation der Knoten beim Programmstart. Damit müssen Anwender über Kenntnisse der Hardware verfügen. Das explizite Angeben von Knontennamen ist aber nur praktikabel für eine überschaubare, d.h. sehr geringe Anzahl von Knoten. Zudem führt diese Methode zu wenig portablen Lösungen.

3.2.2 CHARM++

Charm++ [Kale 93] basiert auf C++. Die Hauptziele in Bezug auf die Parallelisierung sind Portabilität[16] durch die Bereitstellung von Bibliotheksroutinen und die Beseitigung von Blokkierungen infolge blockierendem Sendens und Empfangens durch die Unterstützung asynchroner Kommunikation.

Portabilität wird zusätzlich unterstützt, indem nur wenige Anforderungen an das zugrunde gelegte System gestellt werden. Zu den gestellten Anforderungen gehört vor allem die Möglichkeit dynamischer Prozeßerzeugung.

In Charm++ wird unter Aufgabe der Lokalitätstransparenz im Programm explizit festgelegt, ob ein Zugriff in Bezug auf einen gemeinsamen Adreßraum lokal oder entfernt ist. Zusätzlich werden Prozesse so strukturiert, daß nur Nachrichten versendet werden und die Bearbeitung eines Blocks immer bis zum Ende ausgeführt werden kann, bis auf eine Nachricht gewartet werden muß. Liegt dann keine Nachricht von einem anderen entfernten Prozeß vor, wird dem Prozeß der Prozessor entzogen und an einen weiteren vergeben, für den eine Nachricht gesendet wurde. In Charm++ gibt es daher keine explizite Empfangsprimitive. Die Erweiterung

[16] Charm++ wurde auf mehreren Parallelrechnern implementiert, wie z.B. Rechnern mit gemeinsamen Speicher (nCUBE/2), Einprozessorsystemen (uniprocessors) und Workstation-Clustern. Charm++ läuft ohne Änderungen auf allen MIMD Maschinen [Kale 93].

zum herkömmlichen Vorgehen besteht nun darin, daß ein Prozeß gleichzeitig auf mehrere Nachrichten warten kann.

In Charm++ werden ein oder mehrere Objekte auf einen Prozeß abgebildet. Objekte, die einen gemeinsamen Teil des Gesamtproblems bearbeiten, werden zu sogenannten „chares“ (aktives Objekt) gebündelt und durch einen Prozeß realisiert. Ein „chare“ stellt somit eine mittelgrobe Struktur dar (engl.: medium grain parallelism). Feinkörnige Nebenläufigkeit innerhalb eines Objektes (threads) wird dagegen abgelehnt. Alle Objekte in Charm++ arbeiten intern sequentiell.

Jedes Charm++-Programm muß genau eine chare-Klasse „main“ enthalten. Von dieser Klasse darf es nur ein Objekt geben. Die Programmausführung beginnt mit dem Aufruf dieses Objekts. „main“ dient dazu, dynamisch neue „chares“, „branch-office chares“ (vervielfachte aktive Objekte) und gemeinsame Variablen zu erzeugen und zu initialisieren.

„branch-office chares“ unterstützen reguläre und datenparallele Berechnungen. Sie unterstützen somit das SPMD-Programmiermodell. Wird ein Objekt einer branch-office chare-Klasse erzeugt, wird auf jedem Knoten ein Objekt dieser Klasse erzeugt. Ein vervielfachtes aktives Objekt ist über einen global eindeutigen Namen auf drei unterschiedliche Arten erreichbar. Wird ein solcher Name referenziert mit der Angaben „local“, wird das lokale Objekt aktiviert. Darüber hinaus kann eine Nachricht an einen Vertreter eines vervielfachten aktiven Objekts auf einem bestimmten Knoten durch Angabe der Knotennummer oder an alle (broadcast) geschickt werden. Aktive Objekte und vervielfachte aktive Objekte können dynamisch erzeugt werden, erstere an jede Stelle im Programm, letztere nur in „main“. Aktive Objekte werden nach einer Lastverteilungsstrategie auf Knoten verteilt.

Statt globaler Variablen werden spezifische abstrakte Objekttypen zur Verfügung gestellt, deren Funktion und Schnittstelle genau festgelegt ist, deren eigentlicher Code aber von den Benutzern bereitzustellen ist. Diese Objekte können wiederum nur in dem Hauptprozeßobjekt „main“ erzeugt und initialisiert werden. Zu den gemeinsamen Objekten (engl.: shared objects) gehören folgende:

- Nur-lesbares Objekt (engl.: read-only object): Diese Objekte funktionieren wie nur einmal beschreibbare Variablen.
- Einmal-beschreibbares Objekt (engl.: write-once object): Diese Objekte bekommen erst zu irgendeinem Zeitpunkt während der Laufzeit einen Wert. Nach dieser Initialisierung können sie nur noch gelesen werden.
- Akkumulatorobjekt (engl.: accumulator object): Akkumulatoren dienen der schnellen Aktualisierung von Werten, weshalb sie zwar von allen „chares“ verändert werden dürfen, aber nur einmal gelesen werden können. Ein Akkumulator kann nur mittels zweier (benutzerdefinierter) Funktionen manipuliert werden. Die Funktion „add“ fügt zu einem Objekt etwas hinzu. Die Funktion „combine“ kombiniert zwei Akkumulatorobjekte. Diese beiden Operatoren müssen kommutativ und assoziativ sein.
- Monotones Objekt (engl.: monotonic object): Monotone Objekte werden durch eine idempotente, kommutative und assoziative Operation aktualisiert, deren Schnittstelle vordefiniert ist. Der jeweilige Wert, auf den beliebig oft zugegriffen werden kann, repräsentiert den zu einem Zeitpunkt verfügbaren und nicht den endgültigen Wert. Der verfügbare Wert nähert sich mit der Zeit dem endgültigen immer weiter an.
- Verteilte Tabelle (engl.: distributed table): Verteilte Tabellen bestehen aus einer Menge von Einträgen, welche sich aus einem Schlüssel und einem Datenfeld zusammensetzen. Einträge können mit den nicht-blockierenden Operationen „Insert“, „Delete“ und „Find“ manipuliert werden.

3.2.3 Mentat

Mentat ist ein objektorientiertes paralleles System, welches Bibliotheksroutinen für die Konstruktion portabler, mittelgrober paralleler Programme durch die Kombination eines objektorientierten Ansatzes und eines darunterliegenden virtuellen Maschinenmodells zur Verfügung stellt [Grimshaw 93b]. Eine Implementierung als virtuelle Maschine isoliert Architekturabhängigkeiten und erleichtert somit die Portierung auf andere Maschinen. Mentat liegt das MIMD-Programmiermodell zugrunde.

Mentat erweitert C++ um Mechanismen, welche es Programmierern erlaubt, Anwendungsklassen aufgrund ihres Berechnungsumfangs als eigenständig parallel ausführbare Aktivität zu markieren [Grimshaw 93a]. Der Compiler und das Laufzeitsystem organisieren Kommunikation, Synchronisation und letzteres auch dynamisch das Scheduling. Diese mittelgroben Objekte stellen die Akteure dar. Diese Akteure werden als Knoten innerhalb eines Datenabhängigkeitsgraphen repräsentiert. Die Datenabhängigkeiten zwischen den Knoten werden vom Compiler anhand des Programms erkannt und entsprechend behandelt. Mentat-Konstrukte dienen dazu, das Problemwissen von Programmierern für die Codierung der Granularität, der Dekomposition und Plazierung eines Anwendungsprogramms zu nutzen. Diese werden im folgenden vorgestellt [Grimshaw 92]:

- Mentat: Um dem Compiler zu signalisieren, daß Objekte einer Klasse parallel zu anderen ausgeführt werden sollen, wird das Schlüsselwort Mentat Klassendefinitionen vorangestellt. Die minimale Größe eines so ausgezeichneten Objekts wird mit einigen hundert Instruktionen angegeben [Grimshaw 93a]. Bei einer geringeren Größe verlangsamt der Verwaltungsaufwand zusammen mit der Latenzzeit die Programmausführung anstatt sie zu beschleunigen.
  Objekte, die miteinander kommunizieren sind Mentat-Objekte. Mentat setzt keinen (virtuell) gemeinsamen Adreßraum voraus. Daher werden gemeinsame Daten, z.B. Einträge einer verteilten Datenbank, über Nachrichtenobjekte ausgetauscht.

Mentat-Objekte werden durch Prozesse realisiert. Da sich der Compiler bzw. das Laufzeitsystem aufgrund von Datenabhängigkeitsgraphen um die Synchronisation und Kommunikation kümmert, kann daneben auch Innerobjektparallelität (engl.: intraobject parallelism) transparent abgewickelt werden. Das Schlüsselwort Mentat wird immer in Kombination mit einem der beiden folgenden „Regular“ und „Persistent“ benutzt:

- Regular: So bezeichnete Mentat-Klassen sind zustandslos. Das Laufzeitsystem kann jederzeit ein neues Objekt erzeugen, um einen Methodenaufruf abzuarbeiten.
- Persistent: Diese Klasse von Objekten unterstützen Zustandsinformationen. Ein Objekt einer Persistent_Mentat-Klasse hat einen systemweit eindeutigen Namen und einen unabhängigen Kontrollfaden. Aufrufe werden anhand der Monitoreigenschaft von Objekten synchronisiert.

Der Methodenaufruf ist syntaktisch identisch mit dem in C++. Auf der Ebenen der Semantik ergeben sich zwei wichtige Unterschiede. Zum einem ist jedem Mentat-Objekt ein eigener Adreßraum zugeordnet. Daher werden bei Methodenaufrufen keine Zeiger als Parameter übergeben. Zum anderen wird ein Methodenaufruf in einen nichtblockierenden, d.h. asynchronen RPC (Remote Procedure Call) umgewandelt. Der Aufrufende blockiert erst, wenn das Ergebnis des Aufrufs benötigt wird. Die Synchronisierung übernimmt das Laufzeitsystem anhand eines Datenabhängigkeitsgraphen. Damit wird innerhalb von Mentat Interobjektparallelität transparent unterstützt.

Nachdem Programme auf Anwendungsebenen partitioniert werden, kann auch der Knoten, auf dem ein Objekt letztendlich erzeugt werden soll, näher spezifiziert werden. Dazu wird die Funktion „create()“ mit unterschiedlichen Parametern versehen:

- object.create(): Hier wählt das System den Knoten.
- object.create(COLOCATE another_object): „object“ wird auf dem selben Knoten erzeugt, auf dem sich auch „another_object“ befindet.
- object.create(DISJOINT object1, object2, …): Das Objekte wird auf einem anderen Knoten erzeugt, als die aufgelisteten.
- object.create(HIGH_COMPUTATION_RATIO): Ein Objekt, das wesentlich mehr rechnet als kommuniziert, wird auf einem Knoten erzeugt, der sich durch hohe Kommunikationskosten auszeichnet.
- object.create(int on_host): Ein spezifischer Knoten wird ausgewählt.

Desweiteren können neu zu erzeugende Objekte an bereits existierende gebunden werden:

- object.bind(int scope): Für den Parameter „scope“ können für die Begrenzung einer Objektsuche folgende Werte eingesetzt werden:
  - BIND_LOCAL: auf den lokalen Knoten; In der Mentat-Terminologie kann dies auch ein Mehrprozessorsystem sein.
  - BIND_CLUSTER: auf einen Cluster oder Unternetz und
  - BIND_GLOBAL: auf das Gesamtsystem.

Die wesentlichen Unterschiede zu C++ sind daher:

- Aufrufe sind immer dann nichtblockierend zur Unterstützung von Parallelität, wenn es die Datenabhängigkeiten erlauben. Dies geschieht transparent (interobject parallelism encapsulation).
- Jeder Aufruf eines zustandslosen Objekts erzeugt ein neues Objekt dieser Klasse. Dies erhöht ebenfalls den Parallelisierungsgrad.

Mentat erreicht gute Leistungsergebnisse für viele unterschiedliche Anwendungen auf MIMD-Plattformen, d.h. von lose gekoppelten Arbeitsplatzrechnernetzen bis zu eng gekoppelten Mehrprozessorsystemen. Zusätzlicher Aufwand entsteht während der Ausführung von Mentat-Anwendungen aufgrund der Konstruktion des Programmgraphen, dem Ordnen, Übermitteln und Abgleichen von Argumenten (engl.: argument matching), dem Objekterzeugen und Scheduling .

Mentat nutzt einerseits Anwenderwissen, um Programme zu partitionieren und zu plazieren. Andererseits verlangt dies auch vom Anwender ein tieferes Verständnis der benutzten Hardwarearchitektur. Wie sonst kann ein Programmierer wissen, wie aufwendig eine Instruktionsfolge oder Kommunikation ist. Nach [Kale 93] überlädt Mentat die Konstruktion des Methodenaufrufs und macht es Programmierern gerade unmöglich, die Kosten von Kommunikation abzuschätzen. Zur Partitionierung können Compiler/Parser aber zumindest Vorschläge machen und somit den Programmierer entlasten. Und andererseits könnten die Entscheidungen eines Programmierers auch überprüft werden und gegebenenfalls Gegenvorschläge erstellt werden. Desweiteren stellt Mentat keine speziellen Datenstrukturen zur Programmierung regelmäßiger, datenparalleler Berechnungen bereit.

3.2.4 pC++

P(arallel)C++ unterstützt Datenparallelität durch Sprachkonstrukte zur Zerteilung und Verteilung regelmäßiger Datenstrukturen und Vervielfältigung des Kontrollflusses zur asynchronen

Manipulation dieser Datenfragmente. Diese Mechanismen werden in Form von Basisklassen bereitgestellt. pC++, das auf C++ basiert, realisiert somit ein SPMD-Programmiermodell. Die wesentliche Erweiterungen von pC++ betreffen die Zerteilung und Verteilung der Daten und des Kontrollflusses. Dabei wurde bei allen Erweiterungen auf Konsistenz mit High Performance Fortran [HPF-Forum 93] geachtet.

Große Datenmengen innerhalb eines Objekts werden partitioniert und der dazugehörige Kontrollfuß entsprechend der Anzahl der Partitionen vervielfältigt. Zusätzlich zum datenparallelen Modell kann ein Teil des Programms als ein einziger Kontrollfluß betrachtet werden, der parallele Operationen startet auf strukturierten Mengen verteilter Objekte, die Aggregate genannt werden. Andererseits können diese parallelen Operationen als interagierende Kontrollflüsse betrachtet werden, die über Wissen von Datenlokalität verfügen. Für die Realisierung der Datenparallelität erben Aggregate von der Basisklasse „collection“. Auf Erben der Klasse „collection“ werden die Elemente des Aggregates auf- und verteilt und Methodenaufrufe auf das Aggragat werden parallel auf allen Elemente ausgeführt.

Um Aggregate aber mit identischen Kotrollfäden manipulieren zu können, müssen nun auf allen Knoten die entsprechenden Objekte erzeugt werden. Dazu dient die Anweisung „Processors P“, wobei P die Anzahl der benötigten Prozesse repräsentiert. Mit P als Parameter zur Erzeugung eines collection-Objekts werden dann P Objekte mit der jeweils dazugehörigen Datenpartition und dem Kontrollfaden ausgestattet. Da in der aktuellen Version von pC++ immer alle Knoten in die Verteilung von Datenobjekten einbezogen werden, ist eine geschachtelte Erzeugung von Prozeßobjekten nicht sinnvoll und daher nicht erlaubt.

Da als Zielarchitekturen sowohl Architekturen mit gemeinsamem als auch verteiltem Speicher möglich sind, wird bei letzteren ein gemeinsamer Namensraum für Objekte einer verteilten Datenstruktur durch das Laufzeitsystem unterstützt.

Elemente einer „collection“ werden in zwei Schritten auf die vorhanden Knoten verteilt. Im ersten Schritt werden die Elemente auf ein Template-Objekt abgebildet. Templates sind abstrakte Koordinatensysteme, die es erlauben Elemente verschiedener „collections“ in Bezug aufeinander auf die zur Verfügung stehenden Knoten abzubilden. Darüber hinaus wird ein Template charakterisiert durch die Anzahl seiner Dimensionen, bei Matrizen z.B. durch die Größe jeder Dimension und die Abbildungsvorschrift der Aggregate auf Rechnerknoten.

Im zweiten Schritt wird das Template wird durch einen Abbildungsvorschrift (engl.: distribution) auf die Knoten abgebildet. Bisher realisierte Verteilungen sind BLOCK, CYCLIC, und WHOLE. Durch BLOCK werden die Daten einer Dimension in soviel Blöcke wie Knoten vorhanden sind eingeteilt. Durch CYCLIC werden die Datenpartitionen in einem Round-Robin-Verfahren auf die Knoten abgebildet. Mit WHOLE werden alle Daten auf einen einzigen Knoten verfrachtet.

Die Beendigung der Prozessorelemente muß synchronisiert werden für klassenspezifische Funktionen, da die Bearbeitung der einzelnen Kontrollflüsse asynchron zueinander stattfindet. Eine parallele Ausführung wird mit einer barrier-Synchronisation[17] beendet.

Jede Anwendung hat ihre eigene Kommunikationstopologie und globale Operatoren, welche durch die Struktur der „collection“ unterstützt werden müssen. Zugriff auf entfernte Elemente haben Prozessorobjekte durch in der maschinenabhängigen Klasse „kernel“ definierte Funktionen, z.B. Get_Element(). Die Aufgabe der Kernel-Klasse ist es, den globalen Namensraum für die collection-Elemente zur Verfügung zu stellen. Zu jedem lokalen Objekt einer collection-Klasse wird daher ein Managerprozeß erzeugt, der eine Tabelle mit Einträgen der übrigen

[17] Ein „barrier“ ist gleichbedeutend zu einem „coend“.

Managerprozesse und deren Elemente verwaltet. Ein Zugriff auf ein entferntes Element wird in zwei Schritten durchgeführt:

1: Anfrage an den Managerprozeß, welcher Knoten gerade das gewünschte Element besitzt;
2: Laden einer Kopie des Elements von diesem Knoten auf den eigenen; Dabei ist zu beachten, daß die einzelnen Kontrollfäden asynchron abgearbeitet werden und für die notwendige Synchronisation gesorgt werden muß.

Im Unterschied zum üblichen nachrichtenorientierten Programmieren in der SIMD-Programmierung dürfen alle Prozeßobjekte alle Daten anderer Prozeßobjekte lesen. Aber nur der Besitzer darf seine Daten ändern (engl.: owner computation).

Das Laufzeitsystem von pC++ erfüllt in Bezug auf „collections“ folgende Aufgaben [Bodin 93]:

- Verteilung der collection-Klasse: Durch die Interpretation der „aligment[18]“- und „distribution[19]“ Direktiven werden lokale Datenkollektionen für jedes Prozeßobjekt erstellt.
- Die Verwaltung des Zugriffs auf die Datenelemente, d.h. eine effiziente Implementierung von Kommunikationsoperationen, welche in der Kernel-Klasse definiert sind.
- Die Beendigung der parallelen Operationen auf collection-Objekte, d.h. Durchführung einer barrier-Synchronisationsoperation.

3.2.5 pSather

P(arallel)Sather ist eine Erweiterung der auf Eiffel [Meyer 88] basierenden Programmiersprache Sather um Parallelität. Sather zeichnet sich durch eine einfache Syntax, parametrisierbare Klassen, Mehrfachvererbung und ein strenges Typsystem aus.

Ein Ziel von pSather ist, das Modell des geteilten Speichers sowohl auf Maschinen mit gemeinsamen Speicher (wie z.B. Sequent Symmetry) als auch auf Maschinen mit verteiltem Speicher (wie z.B. CM5) zu unterstützen. Letzteres aufgrund der Skalierbarkeit dieser Architekturen. Um diese beiden Modelle miteinander kombinieren zu können, stellt pSather ein Zwei-Stufen-Modell eines gemeinsamen Speichers (engl.: two-level shared address space) zur Verfügung. Dazu werden Cluster gebildet, wobei jedem Cluster ein (virtuell) gemeinsamer Speicher zugeordnet ist. Für Architekturen mit gemeinsamem Speicher ist die Anzahl der Cluster gleich eins. Bei Architekturen mit verteiltem Speicher werden mehrere Knoten zu einem Cluster bzw. zu einem virtuell gemeinsamen Speicher zusammengefaßt. Die Bildung der Cluster wird aufgrund der Kommunikationseigenschaften zwischen den Knoten optimiert. Im folgenden werden die Eigenschaften von pSather für Architekturen mit verteiltem Speicher betrachtet.

pSather unterstützt sowohl das MIMD- als auch das SPMD-Programmiermodell. Objekte werden entweder durch einen oder mehrere Prozesse realisiert. Objekte können zudem aus vielen dynamisch erzeugten, asynchronen oder synchronen nebenläufigen leichtgewichtigen Prozessen bestehen.

Die Synchronisation von Kontrollstrukturen und der Zugriff auf gemeinsame Daten muß explizit vorgenommen werden. Dazu stehen zwei vordefinierte Klassen GATE0 und GATE{T} mit einer Menge von Attributen und Routinen zur Verfügung. Vordefiniert bedeutet hier, daß die Funktionalitäten der Klassen nicht umdefiniert werden können durch die erbende Klasse. Objekte vom Typ GATE{T} enthalten einen Wert vom Typ T, z.B. einen Zähler. GATE0 ent-

[18] Aligment: Partitionierungsanweisung für ein „collection“-Element.
[19] Distribution: Abbildung eines Templates auf einen Knoten.

spricht einem binärem Semaphore. Diese Konstrukte gewähren den exklusiven Zugriff auf Objekte. Auf der Basis von Gates können komplexere Synchronisationskonstrukte erstellt werden, wie z.B. barrier-Synchronisation oder future-Variablen.

pSather kennt asynchrone Funktionen mit und ohne Rückgabewerte. „GATE0“bzw. „GATE{T} können auf der linken Seite eines nicht-blockierenden Funktionsaufruf stehen (engl.: deferred assigment). Durch dieses Vorgehen wird die jeweilige Funktion synchronisiert. Für asynchrone Funktionen mit Rückgabewert dienen Gates als future-Variablen.

Entfernte Prozeduraufrufe werden in pSather durch Abhören (engl.: polling) angenommen, d.h. ein Prozessor horcht von Zeit zu Zeit auf entfernte Anfragen. Dazu wird er entweder explizit in der Abarbeitung eines Kontrollfadens unterbrochen mittels einer vom Compiler generierten Anweisung oder das Abhören der Leitung wird durchgeführt, wenn es keine bereiten Kontrollfäden bzw. Prozesse gibt.

Solange der Zugriff auf entfernte Daten wesentlich mehr Zeit in Anspruch nimmt als der auf lokale Daten, werden Routinen dort ausgeführt, wo die Daten sich befinden oder die Routinen dorthin kopiert, wo die Daten sich befinden. Dazu bezeichnet der @-Operator den Cluster, in dem die Routine ausgeführt werden soll. Auf diese Weise können sowohl blockierende als auch nicht-blockierende Aufrufe ausgeführt werden.

Das Auffinden von Objekten im zweistufigen Adreßraums ist aufwendig. Objektreferenzen werden dazu gekennzeichnet mit einem Attribut „far“ für Objekte, die sich nicht, und „near“ für solche, die sich im Cluster befinden. Um die Prüfung dieses Attributs einzusparen, kann mit einer with-near-Anweisung die Ausführung eines Kontrollfadens an den lokalen Cluster gebunden werden.

Die Basisklasse SPREAD{T} dient zur Unterstützung von auf alle Cluster zu verteilenden Objekten. Um möglichst effizienten Code zu erhalten, werden alle Partitionen eines verteilten Objekts auf dem jeweiligen Cluster an identischen Adressen plaziert. Durch eine vorgegebenen Broadcast-Routine werden die einzelnen Replikate aktualisiert. (Der Erhalt der Reihenfolge aufeinanderfolgender Broadcasts wird nicht zugesichert.) Durch Vererbung können anwendungsspezifische Klassen definiert werden.

Durch Sprachen wie High Performance Fortran wurde gezeigt, daß die Unterstützung von Datenparallelität durchVerteilung von Feldern zerteilt in Blöcke auf verschiedene Prozessoren von Nutzen ist. Bisher wurde in pSather eine parallele forall-Anweisung implementiert, die keine Garantie über die Reihenfolge der Interaktionen gibt. Diese Anweisung verfügt über Wissen in Bezug auf die Verteilung durch die beiden vordefinierten Klassen DIST_DATA{T} und DIST_ARRAY{T}. Dadurch wird es möglich Iterationen dorthin zu verteilen, wo die Datenstücke lokal sind. Alle nach der Anweisung „dist“ aufgeführten Anweisungen werden parallel auf allen Teilobjekten ausgeführt. „dist“ allein führt nur zu einer Synchronisation des Beginns der vervielfachten Kontrollflüsse. Um auch Berechnungsabschnitte oder die Beendigung zu synchronisieren, kann innerhalb des dist-Blocks die Anweisung „sync“ benutzt werden. „sync“ realisiert eine barrier-Synchronisation. Diese Konstrukte zusammen entsprechen dem SPMD-Programmiermodell.

3.3 Definition einer neuen Sprache

3.3.1 ABCL

Die Sprache ABCL[20] ist beeinflußt durch das Actor-Modell von [Hewitt 77], unterscheidet sich aber vor allem in seiner Behandlung von Nachrichten von diesem. Innerhalb des Actor-Modells wird für die Behandlung einer Nachricht jeweils ein neuer Akteur erzeugt. In ABCL behalten aktive Objekte Zustände auch über die Bearbeitung einer Nachricht hinaus bei.

Programme in ABCL bestehen aus Objekten, welche durch Erhalten einer Nachricht aktiv werden. Es können mehr als eines dieser Objekte zu einen Zeitpunkt aktiv sein und mehrere Nachrichtenübertragungen können parallel stattfinden. Jedes Objekt hat seinen einzelnen autonomen Kontrollfaden und eine Menge von Zustandsvariablen. Ein Objekt wird durch genau einen Prozeß realisiert.

Statt der Vererbung wird das auf [Lieberman 86] zugrückgehende Konzept der Delegation favorisiert. Dabei sendet (delegiert) ein Objekt A im einfachsten Falle eine von ihm nicht bearbeitbare Nachricht an ein dafür vorgesehenes Objekt B. Objekt B sendet die Antwort direkt an das sendende Objekt zurück, wobei es als Absender das Objekt A angibt, so daß die Delegation transparent bleibt [Yonezawa 90].

In ABCL/M werden Nachrichten selektiv empfangen und synchron oder asynchron versendet. Diese können eine Markierung (engl.: message tags), den Namen weiterer Objekte (ID oder globale Adresse) und Basiswerte, wie z.B. Zahlen oder Wahrheitswerte, enthalten. Ob ein Objekt eine Nachricht akzeptieren kann, wird durch die Markierung, den Wert der Nachricht und den aktuellen Status bzw. Modus des Objekts bestimmt. Es gibt drei Modi: schlafend, wartend und aktiv. Ein Objekt kann eine Nachricht nur akzeptieren, wenn es schläft oder wartet. Ein schlafendes Objekt prüft die eingetroffenen Nachrichten der Reihe nach bis eine Nachricht gefunden wird, die eine bestimmte Bedingung erfüllt. Das schlafende Objekt wird dann aktiv und bearbeitet die Nachricht. Die geprüften Nachrichten, die die Bedingung nicht erfüllen, werden gelöscht. Wartende Objekte durchsuchen den Nachrichtenpuffer nach einer erwarteten Nachricht. Im Unterschied zum schlafenden Zustand werden geprüfte Nachrichten nicht gelöscht.

Versendet ein Objekt eine Nachricht vom Typ „Now“, wartet das Objekt solange bis eine Antwort eintrifft (synchrone Kommunikation), wobei diese nicht vom ursprünglichen Empfänger stammen muß (Delegation).

Asynchrone Nachrichten sind vom Typ „Past“ oder „Future“. Versendet ein Objekt eine Nachricht vom Past-Typ, setzt es seine Arbeit unmittelbar fort. Eine (Rück-) Nachricht wird nicht erwartet. Wird eine Nachricht vom zweiten Typ versendet, setzt das Objekt die Bearbeitung fort, bis es die Antwort benötigt. Für die zu einem späteren Zeitpunkt erwartete Antwort stehen sogenannte future-Objekte zur Verfügung.

Der ABCL-Dialekt ABCL/1 kann darüber hinaus mittels express-Nachrichten beim Empfänger eine Unterbrechung auslösen und diesen zur Durchführung einer dringenden Aufgabe zwingen [Yonezawa 86].

Die Definition eines Objektverhaltens muß festlegen, wie sein lokaler Speicher repräsentiert wird, welche Nachricht unter welcher Bedingung akzeptiert oder an welches Objekt delegiert wird und welche Methode gegebenenfalls abzuarbeiten ist. Abgesehen von der Parallelität/-

[20] ABCL/M: An Object-Based Concurrent Language/Model

Nebenläufigkeit sind ABCL-Objekte den Objekten sequentieller Programmiersprachen wie Smalltalk und C++ ähnlich.

3.3.2 Concurrent Aggregates (CA)

CA wurde beeinflußt durch das Actor-Modell und Concurrent Smalltalk. Sein Objektmodell basiert auf dem Actor-Modell, d.h. jedes Objekt kann immer nur eine Nachricht zu einer Zeit bearbeiten. Der Basisadressierungsmechanismus „one-to-one-of-many“ für Aggregate (siehe unten) und die Möglichkeit, innerhalb eines Aggregates andere Akteure zu adressieren, ähneln nebenläufigen Konstrukten von Concurrent Smalltalk. CA ist insofern eine neue Sprache in Bezug auf das Actor-Modell, als ein neues Adressierungsschema zugrundegelegt wird [Chien 93].

CA wurde für die Programmierung spezieller feinkörniger Computer, wie etwa der J-Machine [Dally 89] entworfen. Diese Rechner zeichnen sich durch massive Parallelität aus, d.h. sie verfügen über mehr als 10^5 Knoten, schnelles Kommunikationsnetz und wenig lokalen Speicher[21]. Die Hardware unterstützt feinkörnige Strukturen durch sehr schnelle Nachrichtenübertragung, schnellen Kontextwechsel und effiziente Prozeßerzeugung. CA basiert auf dem MIMD-Programmiermodell mit der Möglichkeit, datenparallele Probleme adäquat zu beschreiben.

Elemente von CA sind Objekte, Aggregate, Nachrichten und „continuations“. Diese werden im folgenden näher erläutert.

Ein Objekt wird definiert durch eine Klassen-, Methoden- und Delegationsdeklaration. Sein lokaler Zustand enthält unter anderem die Namen/Adressen anderer Objekte zu Kommunikationszwecken. Ein Objekt wird durch einen Prozeß realisert. Daneben können die einzelnen Methoden eines Objekts nebenläufig zueinander ausgeführt werden. Dies muß bei der Methodendefinition durch das Schlüsselwort „no_exclusion“ explizit angegeben werden. Ansonsten werden die Methodenausführungen automatisch synchronisiert. Innerhalb von Anweisungen gibt es die Möglichkeit zwischen nebenläufiger (conc oder forall) und sequentieller Ausführung zu unterscheiden. Die nebenläufige Ausführung wird nur durch Datenabhängigkeiten eingeschränkt. Durch den Operator „forall“ werden Schleifendurchläufe nebenläufig zueinander ausgeführt.

Ein Aggregat ist eine Menge von Repräsentanten (engl.: representatives), welche mit einem Namen referenziert werden können [Chien 93]. Ein Repräsentant ist ein Objekt bzw. ein Akteur. Ein Aggregat kann von außen als ein einziges Objekt betrachtet werden. Dies dient der Vereinfachung der Programmierung. Andererseits erhöht sich der Parallelität durch die Vielzahl der Repräsentanten, deren Anzahl bei der Erzeugung individuell festgelegt wird. Nachrichten an ein Aggregat werden einem vom Laufzeitsystem willkürlich ausgewählten Repräsentanten zugeordnet (one-to-one-of-many-Mechanismus). Ein Akteur eines Aggregates kann auf eine Nachricht reagieren, indem er:

- eine Nachricht an einen anderen Akteur oder ein anderes Aggregat schickt,
- die entsprechende Berechnung durchführt oder
- einen neuen Akteur erzeugt.

[21] Es kann einen globalen Speicher geben, aber es wird ein Architekturmodell mit verteiltem Speicher zugrundegelegt.

Ein Aggregat wird durch eine Aggregats-, eine „handler"- und eine (optionale) Delegationsdeklaration definiert. Eine handler-Deklaration beschreibt das Verhalten eines Aggregatsrepräsentanten. Durch das Einführen von Abstraktionshierarchien wird die Komplexität der parallelen Programmierung verringert ohne dabei die Parallelität einzuschränken [Chien 90].

Um parallelisierbare Aufgabenstellungen möglichst effizient abarbeiten zu können, ist es mittels aggregatsinterner Adressierung erlaubt, den Namen (Index) eines jeden Repräsentanten zu bestimmen. Damit wird Kommunikation und Kooperation innerhalb eines Aggregats ermöglicht. Für effiziente Kommunikation ist es darüber hinaus möglich, Repräsentanten aus anderen Aggregaten zu referenzieren. Die Ergebnisse der einzelnen Repräsentanten müssen allerdings durch explizites Nachrichtenversenden akkumuliert werden (siehe auch Charm++).

Nachrichten lösen Berechnungsschritte aus und koordinieren diese. Nachrichten sind asynchron. Für Antworten werden sogenannte „continuations" (gleiche Funktionsweise wie future-Variablen) bereitgestellt. Eine „continuation" kann nur einmal beschrieben und gelesen werden. Mit Hilfe dieses Konstrukts können weitere Synchronisationsmechanismen implementiert werden, wie z.B. „barriers".

3.3.3 Ellie

Ellie ist eine feinkörnige objektorientierte Sprache, die vor allem das MIMD-Programmiermodell unterstützt.

Um möglichst alle Granularitätsstufen von Parallelität abdecken zu können, unterstützt Ellie [Andersen 92a, 92b] sehr feinkörnige Objekte. Feinkörnige Objekte sind z.B. natürliche Zahlen oder Zeichen und deren Methoden. Gröber gekörnte Objekte, d.h. Objekte, die mehrere feinkörnige Objekte umfassen, werden Ellie-Objekte genannt. Aufgerufene Objekte werden auf Prozesse abgebildet.

Die Körnigkeit der Prozeßobjekte wird letztendlich vom Compiler bestimmt. Dabei werden feinkörnige Objekte zu größeren Einheiten zusammengefaßt. Bei jeder Vergröberung der Granularität verringert sich der Verwaltungsaufwand und erhöht sich folglich der Effizienzgewinn. Dieser Vorgang kann solange wiederholt werden bis eine optimale Granularität gefunden wurde.

Mittels Referenzen wird der Zugriff auf entfernte Objekte gewährt. Einerseits können Referenzen zwischen Objekten ausgetauscht werden, andererseits werden Objekte mittels Referenzen an Methoden weitergegeben. Spezialisierungen ergeben sich mittels Delegation bzw. Vererbung.

Ellie stellt als alleiniges Synchronisationsinstrument sogenannte dynamische Schnittstellen (engl.: dynamic interfaces) zur Verfügung. Eine solche Schnittstelle kann die Existenz einer Untermenge von Methoden temporär unerreichbar machen. Aufrufe verborgener Methoden werden in Warteschlangen eingeordnet und später der Reihe nach abgearbeitet.

Die Parallelität kann durch die Unterscheidung von synchroner und asynchroner Kommunikation erhöht werden. Dazu werden Methodenaufrufe durch die Schlüsselworte „bound" für synchrone und „unbound" für asynchrone entfernte Prozeduraufrufe (RPC bzw. URPC) gekennzeichnet. Bei einem entfernten ungebundenen Aufruf wird das Ergebnis an eine „future variable" zurückgeliefert.

Dynamische Lastverteilungsstrategien greifen für Ellie-Objekte kaum, da die Lebensdauer feinkörniger Prozesse kürzer ist als die Aktualisierungszyklen. Daher wird vom Compiler Information zur Lastverteilung in einer Objektstruktur „type object" plaziert. Das größte Problem ist

herauszufinden, welche Information für den Compiler zu diesem Zweck nützlich ist. Programmierer können ihr Wissen in Form von Annotationen bereitstellen, z.B. wo ein Prozeßobjekt erzeugt werden soll, welcher Objekttyp migriert werden soll oder wann eine Umwandlung eines asynchronen in einen synchronen RPC sinnvoll ist [Andersen 92c].

Die Hauptvorteile der Feinkörnigkeit sind nach [Andersen 93]:
- Vereinfachung der Programmierung durch Bereitstellung eines einheitlichen Modells;
- Konformität mit real existierenden Objekten;
- Verschmelzung von Objekten zur Übersetzungszeit[22].

Und deren Hauptnachteile sind:
- der Aufwand beim Erzeugen von feinkörnigen Objekten ist nicht viel geringer als der für gröber gekörnte Objektstrukturen;
- ebenso unterscheidet sich der Aufwand bei der Objektübertragung kaum bei grob und feinkörnigen Objekten[23];
- hoher Speicherplatzbedarf.

[22] Das Aufsplitten von grobkörnigen Objekten führt nicht unbedingt zu optimalen feinkörnigen Programmen auf feinkörnigen Rechnern.
[23] Es ist nicht akzeptabel, daß feinkörnige Objekte eine längere Zeit zum Migrieren in Anspruch nehmen als ihre Lebensdauer beträgt. Solche Objekte sollten mit anderen zu einem Cluster verschmolzen werden und so migriert werden.

4 Gegenüberstellung paralleler objektorientierter Sprachen

Im folgenden werden die im Kapitel 3 beschriebenen Sprachen nach der Kriterien der Taxonomie, wie sie in Kapitel 2 beschrieben wurde, einander gegenübergestellt. Abschließend wird in Kapitel 5 eine kurze Bewertung gegeben.

4.1 Programmiermodelle

Um größere Flexibilität in Bezug auf die Problemklassen zu erhalten, unterstützen viele parallele objektorientierte Sprachen beide Programmiermodelle. Im allgemeinen wird ein Programmiermodell dennoch bevorzugt. CA unterstützt das SPMD-Modell besser als das MIMD-Modell. CC++, Charm++ und pSather unterstützen dagegen das MIMD-Modell mehr. Dennoch unterstützen ein paar Sprachen nur ein Modell. Die folgende Tabelle zeigt einen Überblick über die jeweils unterstützten Programmiermodelle.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
SPMD	-	xx	x	x	-	xx	-	xx	x
MIMD	xx	x	xx	xx	xx	-	xx	-	xx

Tabelle 2: Programmiermodelle

4.2 Nebenläufigkeitsmodelle

Die Granularität der parallel bzw. nebenläufig zueinander bearbeitbaren Aktivitäten ist in den untersuchten Sprachen sehr unterschiedlich (siehe Tabelle 3). Vor allem die Größe der Prozesse unterscheidet sich erheblich. Prozesse in Sprachen wie ABCL, Charm++, Mentat und pSather sind eher umfangreich im Gegensatz zu feinkörnigen Prozessen wie in CA, Ellie und pC++. Außer bei Ellie liegt das am unterstützten Programmiermodell SPMD. Da die Datenmenge datenparalleler Programmsequenzen auf die zur Verfügung stehenden Knoten verteilt wird, ist die Granularität von deren Anzahl abhängig. Insofern schwankt die Granularität der Prozesse je nach Umfang der Datenmengen und der Knotenanzahl. Daher läßt sich über die Prozeßgröße in SPMD-Sprachen, wie beispielsweise in EPEE, nur wenig aussagen.

Die feinkörnige Struktur von CA wird von der Hardware direkt unterstützt. Damit werden Effizienzverluste vermieden. In Ellie ist die Feinkörnigkeit eher abstrakt, da die Objekte vom Compiler zu größeren, den Verwaltungsaufwand reduzierenden, Einheiten zusammengefaßt werden.

Der Grad der Nebenläufigkeit kann durch leichtgewichtige Prozesse erhöht werden. Dies gilt umsomehr, wenn deren Erzeugung und Verwaltung durch die Hardware unterstützt wird. Sowohl in C++ als auch in Mentat geschieht die Erzeugung solcher Prozesse implizit anhand

der vom Compiler gewonnenen Erkenntnisse über die Datenunabhängigkeit einzelner Programmteile.

Der Grad der Nebenläufigkeit läßt sich durch eine Schleifenparallelisierung im allgemeinen nur wenig erhöhen, außer es handelt sich um Schleifen, die auf voneinander unabhängigen Datensegmenten operieren, wie dies beispielsweise bei der Vektormultiplikation der Fall ist. Können Anweisungen parallel oder nebenläufig zueinander ausgeführt werden, muß bei möglichen Datenabhängigkeiten das Laufzeitsystem deren Analyse effizient unterstützen. Sprachen, die eine parallele Abarbeitung von einzelnen Schleifendurchläufen ermöglichen, sind: CA, CC++, Ellie, Mentat und pC++.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
Prozeß	x	x	x	x	x	x	x	x	x
leichtgewichtige Prozesse	-	x (no_exclusion)	x	-	-	-	x	-	x
Anweisungen	-	x	x	-	x	-	x	-	-

Tabelle 3: Granularität der paralleler bzw. nebenläufiger Aktivitäten

Feinkörnige Strukturen nutzen die Möglichkeiten der Parallelisierung eines Problems besser aus. Andererseits behindert die Feinkörnigkeit der Strukturen in gewisser Weise das Zusammenfassen von größeren semantischen Einheiten. Dazu müssen dann entsprechende Konstrukte zur Verfügung gestellt werden, wie z.B. Aggregates in CA.

Wesentlicher als die Prozeßgröße selbst erscheint der vom Betriebsystem zu leistende Aufwand bei der Prozeßerzeugung, die Vermeidung von Kommunikation durch geschicktes Plazieren und ein effizientes Laufzeitsystem.

4.2.1 Synchronisation

Wie in Kapitel 2.2.1 ausgeführt, kann zwischen der Synchronisation zur Erhaltung der Datenkonistenz und der zur Einhaltung eines semantisch korrekten Kontrollflusses unterschieden werden. Im folgenden wird zuerst die Synchronisation des Datenzugriffs ohne Kommunikation besprochen.

Um den Parallelisierungsgrad zu erhöhen, ist eine differenzierte Synchronisation anzustreben. Dies ist aber auch mit dem entsprechendem Aufwand verbunden, z.B. dem Erlernen des richtigen Einsatzes unterschiedlicher Mechanismen oder einem aufwendigeren Compiler. Eine automatisierte Auswahl der jeweils minimal notwendigen Synchronisation hätte eine weitere Vereinfachung der Programmierung und der Programmiersprachen zur Folge.

Je mehr parallele Aktivitäten auf der jeweiligen Ebene zugelassen sind, desto aufwendiger ist die Synchronisation möglicher Datenabhängigkeiten. Synchronisation schränkt die Parallelität ein, und der Aufwand, der für Synchronisation getrieben werden muß, vermindert die Effizienzsteigerung der Parallelisierung. Andererseits gilt, daß je umfangreicher die Objekte sind, desto mehr nebenläufige Aktivitäten möglich sein sollten, um die Rechnerleistung möglichst voll auszuschöpfen. Es sollten z.B. gleichzeitiges Lesen oder mehrere nebenläufige Aufrufe verschiedener Methoden eines Objekts möglich sein. Dies gilt vor allem für MIMD-Sprachen.

Bei SPMD-Sprachen verringert sich der Synchronisationsaufwand allein schon durch deren relativ große Datenunabhängigkeit. Letztendlich wird es auf einen Kompromiß zwischen den beiden Anforderungen hinauslaufen, vor allem wenn dies auf der Ebene der Programmiersprache behandelt wird. In diesem Fall wird man zu möglichst flexiblen Konzepten tendieren, um möglichst viel Problemklassen behandeln zu können. Aber angesichts der Komplexität paralleler Ausführung ist es jedoch ungewiß, ob dies immer den erwünschten Effekt hat. Folglich sollte die Synchronisation von Compilern optimiert werden.

Im folgenden werden nur solche Synchronisationsmechanismen erörtert, die nicht in Zusammenhang mit Kommunikation stehen, d.h. ohne die Übermittlung von Nutzdaten. Die Synchronisationsmechanismen mit Kommunikation der Sprachen, wie ABCL oder Charm++ werden im nächsten Abschnitt behandelt. Tabelle 4 gibt einen Überblick über die bereitgestellten Synchroniationsmechanismen zur Erhaltung der Datenkonistenz.

Auf der Ebene der Prozesse kann man diejenigen Sprachen unterscheiden, die entweder nur einen Methodenaufruf pro Objekt zu einer Zeit und den anderen, die mehrere gestatten (siehe Tabelle 4). Erstere nutzten die aus der Objektorientierung stammende Datenkapselung aus. Die anderen verfolgen unterschiedliche Strategien, da zwischen lesenden und schreibenden Zugriff unterschieden werden kann. Ein schreibender Zugriff wird synchronisiert entweder durch Monitore, durch Datenkappselung oder nur einmal schreibbare Variablen.

Bei der Bereitstellung von nur einmal zu schreibenden Variablen wird die Synchronisation durch die Deklaration des Variablentyps festgelegt. Diese Präzisierung einer Eigenschaft unterstreicht den Charakter der Objektorientierung stärker, als die Bereitstellung von Monitoren oder ähnlichem, das eher auf Eigenschaften des Betriebsystems verweist.

Auf der Ebene der leichtgewichtigen Prozesse werden im Prinzip die gleichen Strategien verwendet. Hinzukommen kann die Synchronisation durch eine Datenabhängigkeitsanalyse durch das Laufzeitsystem.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
Monitor usw.	-	-	-	x	-	-	-	-	x (gates)
Daten-kapselung	x	x	x	x	x	-	x	-	-
einmal beschreib-bare Variablen	-	x	x	x	-	-	-	-	-

Tabelle 4: Synchronisationsmechanismen zur Erhaltung der Datenkonsistenz

Neben der Synchronisation zur Konsistenzerhaltung von Daten müssen Sprachen auch Hilfsmittel zur Einhaltung des semantisch korrekten Kontrollfusses bereitstellen, um die Reihenfolge der Ausführung von Aktivitäten zu synchronisieren.

Die Reihenfolge der Ausführung von Anweisungen kann implizit mit Hilfe einer Datenabhängigkeitsanalyse synchronisiert werden, d.h. eine Anweisung wird solange verzögert bis eine andere benötigte Daten zur Verfügung gestellt hat (z.B. Mentat). Die Synchronisation von Blöcken oder Schleifen kann durch die Semantik von Sprachkonstrukten gegeben sein, wie z.B. in CC++ durch „par“ und „parfor“, und durch den Compiler automatisch parallelisiert werden.

Gröbere Aktivitäten werden im allgemeinen explizit durch Sprachkonstrukte synchronisiert. Die Synchronisation von Aktivitäten kann sich dabei auf deren Beginn, auf deren Ende oder auf beides beziehen.

Das („klassische“) SPMD-Modell erfordert einen definierten Zeitpunkt der Beginns und der Beendigung aller datenparallelen Programmsequenzen. Diesem tragen Sprachen Rechnung, beispielsweise durch die Bereitstellung von barrier-Synchronisation. Diese Synchronisationsart kann auch zur alleinigen Endesynchronisation verwendet werden (siehe Tabelle 5). Während oder nach Beendigung eines Aktivitätszyklus kann durchaus Kommunikation zwischen Knoten stattfinden.

Über von EPEE unterstützte Synchronisationsmethoden können Aussagen nur aufgrund des unterstützten Programmiermodell getroffen werden. In konkreten Implementierungen können durchaus weitere Mechanismen vorhanden sein.

Sprachen, die das MIMD-Modell unterstützen, synchronisieren die Beendigung asynchroner Aktivitäten auf Prozeßebene ebenfalls durch „barrier“, wie z.B. in CA und Mentat. Auffallend ist, daß Konstrukte wie z.B. Aggregate in CA, die der Steigerung der Parallelität dienen, nicht mit dem Austausch von Nutzdaten synchronisiert werden.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
nur festgelegter Beginn	-	x (aggregates)	-	x (branch-office chare zus broadcast)	-	-	-	-	x (dist ohne sync)
nur festgelegtes Ende	-	x (barrier)	-	x (accumulator)	-	-	x (barrier)	-	-
festgel. Beginn und Ende	-	x (barrier)	x (par; parfor)	x	-	x	x (barrier)	x (barrier)	x (dist zus. sync)
Datenabhängigkeitsana.	-	-	x	-	-	-	x	-	-

Tabelle 5: Synchronisationsmechanismen zur Erhaltung des korrekten Kontrollflusses

Sprachen, die mehr als ein Programmierparadigma unterstützen und damit mehrere Synchronisationsmethoden benötigen, können für mehr Problemklassen benutzt werden, die unterschiedlichen Teilprobleme adäquater ausdrücken und die Möglichkeiten zur Parallelisierung besser nutzen. Andererseits werden sie aber auch unübersichtlicher.

4.2.2 Kommunikation

Kommunikation dient zum einen dem Austausch von Daten und zum anderen der Synchronisation. Diese Verknüpfung ist am offensichtlichsten bei der synchronen Kommunikation. Diese Kommunikationsart schränkt die Parallelität aber auch am meisten ein. Am flexibelsten ist die Kommunikation, bei der der Zeitpunkt des Eintreffens der Antwort für die Fortsetzung des

Senders nicht von Bedeutung ist, wie in Charm++ oder pSather. In Charm++ gibt es gar keine explizites Empfangen.

Einen Mittelweg beschreiten Sprachen mit asynchroner Kommunikation in Verbindung mit sogenannten future-Variablen. Hier wird die Kommunikation erst einmal von der Synchronisation entkoppelt, um den Sender mit dem Empfänger zu einem späteren Zeitpunkt zu synchronisieren. Dieser Synchronisationspunkt ist spätestens dann erreicht, wenn der Sender die Antwort zur Weiterarbeit benötigt.

Da unterschiedliche Problemstellungen unterschiedlich restriktive Kommunikationsmechanismen erfordern, ist die Bereitstellung von mehreren Mechanismen eine Möglichkeit die ganze Bandbreite der Synchronisationsarten abzudecken (siehe Tabelle 6).

Die Möglichkeit, eine synchrone Kommunikation erst nach einem Test einzugehen, wird von keiner der untersuchten Sprachen realisiert.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
synchron	x (Now)	-	x	-	x (bound RPC)	x	-	x	x
asynchron mit future-V.	x (future)	x (continu-ation)	-	-	x (unbound RPC)	-	x	-	x
asynchron ohne future-V.	x (Past)	-	-	x	-	-	-	-	x

Tabelle 6: Kommunikationsarten

4.3 Zerteilung und Verteilung

Das Zerlegen von Algorithmen in Klassen, Methoden, Schleifen oder Anweisungen, wie sie durch Sprachkonstrukte definiert werden, ist Voraussetzung für die Spezifikation von Aktivitäten, welcher Granularität auch immer. Welche dieser Sprachkonstrukte zu Aktivitäten werden können, hängt vom Ausführungsmodell einer Sprache ab.

Das Ausführungsmodell legt fest, ob Objekte aktiv oder passiv sind, ob leichtgewichtige Prozesse unterstützt werden usw. Dabei sagt die Granularität von Aktivitäten nicht unbedingt etwas aus über der Effizienzgewinn durch die letztendlich erreichte Parallelisierung.

Nach [Giloi 93] erhält man durch die Parallelisierung von Prozessen, leichtgewichtigen Prozessen und Schleifen einen hohen Parallelisierungsgrad. Ob die Kombination von Aktivitäten unterschiedlicher Granularität zu einer Steigerung der Leistung führt und von welchen Bedingungen dies abhängt, muß noch untersucht werden.

Je nach Anwendung wird sich das Problem anders stellen. Es sollte auch untersucht werden, ob sich Unterschiede ergeben zwischen Anwendungen, deren Problemraum sich dynamisch verändert, und Anwendungen mit statischem Problemraum.

Die Abbildung von (aktiven) Objekten auf Prozesse bzw. von Prozessen auf (passive) Objekte ist ein grundlegendes Mittel zur Steigerung der Parallelität. Die Herangehensweise von

Ellie die Abbildung von Objekten auf Prozesse durch den Compiler ausführen zu lassen ist die flexibelste Methode. Die Güte der Abbildung hängt dann von der Qualität des Compilers ab.

Die Möglichkeit ein Objekt als Prozeß zu realisieren ist in all denjenigen Sprachen gegeben, deren Objekte aktiv werden können (1 (Objekt) : 1 (Prozeß)), d.h. in allen bis auf pSather. pSather geht von einem passiven Objektmodell aus.

Die jeweilige Prozeßgröße unterscheidet sich sehr. In Charm++ wird die Bearbeitung eines Teilproblems zu einem Prozeß gebündelt. Der relativ große Umfang von Mentat-Objekten wird durch die Parallelität innerhalb des Objekts ausgeglichen (1:n). Die Größe der Prozesse dürfte sich im allgemeinen nicht als Nachteil erweisen, da dadurch Daten lokal zu einem Prozeß gehalten werden können und damit sich die Kommunikationshäufigkeit vermindert. Die Größe eines Prozesses ist dann von Nachteil, wenn durch das Fehlen einer Ressource der gesamte Prozeß blockiert ist und als Folge keine sinnvolle Arbeit von einem Knoten geleistet werden kann und dieser Verlust an Leistung durch eine kleinteiligere Strukturierung nicht eintreten würde.

Gerade in Sprachen, die SPMD-Probleme effizient unterstützen sollen, spielen flexible Objektaufteilungen (1:N) eine wichtige Rolle. Dies spiegelt sich besonders in den Sprachen EPEE und pC++ wieder, die beide nur dieses Modell unterstützen. CA gehört hier nicht ausschließlich dazu, da die Vervielfältigung der Objekte nicht allein für die Bearbeitung datenparalleler Sequenzen konzipiert ist.

Objekte, die keinen Zustand haben und deren Methoden bei Bedarf jederzeit auf jedem Knoten verfügbar sind, erhöhen die Parallelität. Charm++, Mentat und zum Teil auch CA verfolgen diesen Ansatz. Dieser Konstruktion ist durchweg zuzustimmen.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
1:1	x (actor)	x	x	x	x	x	x	x	x (passiv)
1:n	-	x	-	-	-	-	x (intra object paral.)	-	x
1:N	-	x (aggre-gate)	-	x (branch-office chare)	-	x (dist._ aggre-gate)	-	x (collec-tion)	x (Dist_ Array)
i:1	-	-	-	-	x	-	-	-	-

Tabelle 7: Abbildung Objekt auf Prozeß

Allein mit der Abbildung von Objekten auf Prozesse ist es noch nicht getan. Die so konzipierten Prozesse müssen auch auf den Knoten plaziert werden, entweder statisch zu Beginn einer Berechnung oder dynamisch während der Laufzeit, wenn neue Prozesse erzeugt werden.

Bei SPMD-Sprachen ist dies trivial, da die erzeugten datenparallelen Programmsequenzen auf alle Knoten verteilt werden. Hier wird entweder die datenparallele Berechnung in so viele Teile wie Knoten vorhanden zerlegt und dann auf jedem Knoten ein Prozeß plaziert oder wenn es mehr Prozesse gibt als Knoten nach dem Modulo-Verfahren die entsprechende Anzahl von Prozessen auf einen Knoten plaziert. Schwierig hingegen wird es vor allem bei heterogenen Prozessen in speicherverteilten Systemen. Die Verteilung von Prozessen kann entweder statisch nach Anweisungen des Compilers oder dynamisch zur Laufzeit durch Strategien der Lastver-

teilung erfolgen, beispielsweise wird ein neuer Prozeß auf dem jeweils unausgelastetsten Knoten des Systems gestartet. Lastverteilungsstrategien betrachten im allgemeinen nur die Knotenauslastung zu einem bestimmten Zeitpunkt.

Eine Möglichkeit zur Minimierung der Latenzzeit besteht neben der Verwendung noch schnellerer Kommunikationsmedien darin, das zu erwartende Kommunikationsverhalten von neuen Prozessen bei der Verteilung zu berücksichtigen. Das Kommunikationsverhalten müßte eigentlich durch einen Compiler ergründet werden können. Ist dies der Fall, könnte die Verteilung von Prozessen von diesem Wissen profitieren, indem die Verteilung durch Compilerdirektiven gesteuert wird. Dieser Weg wird aber in keiner der untersuchten Sprachen beschritten.

ABCL und CA sind verteilungstransparent, d.h. auf Sprachebenen kann auf die Verteilung keinen Einfluß genommen werden. Die Sprachen CC++, Charm++, Ellie und pSather unterscheiden sich vor allem darin, wieweit sie die Verteilungstransparenz aufgeben.

Bei zwei der untersuchten Sprachen, Ellie und Mentat, kann der Benutzer das Kommunikationsverhalten näher spezifizieren. Charm++ tut dies indirekt durch die Klassifizierung von Objekten als „chare“ bzw. als „branch-office chare“. Charm++ läßt insofern die Spezifikation von „zueinander entfernten“ Prozessen zu. Ob diese tatsächlich auf unterschiedlichen Knoten plaziert sind, hängt von der jeweiligen Last und der Lastverteilungsstrategie ab.

Da in Ellie Lastausgleichsstrategien aufgrund der geringen Größe der Prozesse nicht greifen, werden Plazierungen mit Hilfe von Anmerkungen gesteuert. Die Anmerkungen steuern die Plazierung relativ zu anderen Prozessen, werden infolgedessen vom Compiler oder Laufzeitsystem in konkrete Knoten umgesetzt. Dieses Vorgehen setzt insofern keine Kenntnisse der verwendeten Hardware voraus. Es schränkt damit auch die Portabilität nicht ein, wie die Sprache CC++ (siehe Tabelle 8), die ganz konkrete Knotenbezeichnungen in ihre Verteilungsstrategie aufnimmt.

	ABCL	CA	CC++	Charm++	Ellie	EPEE	Mentat	pC++	pSather
bestimmter Knoten	-	-	x	-	x	-	x	-	-
alle Knoten	-	-	-	x	-	x	-	x	x
Kommunikationsverhalten	-	-	-	-	x	-	x	-	-
Entfernungsrelationen	-	-	-	-	x	-	x	-	x
Lastverteilung/-System	x	x	x	x	-	-	x	x	x

Tabelle 8: Verteilungsstrategien

5 Zusammenfassung

Bei den untersuchten Sprachen ist eine Tendenz zu einer Auffächerung der Modelle erkennbar. Es werden sowohl MIMD- als auch SPMD-Probleme unterstützende Konstrukte integriert. Dies macht diese Sprachen zuweilen unübersichtlich, aber auch flexibler einsetzbar, d.h. mit einer Programmiersprache können Problemen mehrerer Problemklassen gelöst werden. Besonders interessant ist dies für Probleme, die sowohl MIMD- als auch SIMD-Teilprobleme enthalten.

Sprachen können sowohl für Systeme mit gemeinsamem als auch verteiltem Speicher konzipiert werden. Die sich daraus ergebenden Probleme sollten nicht auf der Sprachebene sichtbar sein (wie z.B. in pSather). Sprachen, die in solchen Fällen die Abstraktionsebenen des virtuell gemeinsamen Speichers oder die des gemeinsamen Namensraum unterstützen, sind daher vorzuziehen.

Die asynchrone Kommunikation erhöht die Parallelität. Neuere Sprachen legen sich nicht mehr fest auf synchrone oder asynchrone Kommunikation. Sie bieten meist verschiedene Möglichkeiten an. Die dadurch gewonnenen Flexibilität ist durchaus zu begrüßen.

Auf der Sprachebene sollten eigentlich die Gegebenheiten der Hardware nicht weiter zum Tragen kommen. Bei parallelen objektorientierten Sprachen reduziert sich dieser Anspruch, je mehr Anteil an effizienter Zerteilung und Verteilung nach semantischen Gesichtspunkten Programmierern überlassen wird. Konkrete Zerteilungs- und Verteilungsangaben, die eine vertiefte Kenntnis der Hardware voraussetzen, sind trotz der dadurch zu erzielenden Effizienzgewinne nicht zu befürworten. Dies sollte Aufgabe des Übersetzers und des Laufzeitsystems sein. Es scheint zumindest so, als würden nicht alle Möglichkeiten des Übersetzens ausgeschöpft. Dies sollte näher untersucht werden.

Eine weitere Flexibilisierung der Parallelität geht mit der Koexistenz von Aktivitäten unterschiedlicher Granularität einher. Inwieweit sie sich gegenseitig unterstützen und zu einer weiteren Leistungssteigerung führen, sollte ebenfalls näher untersucht werden.

6 Literatur

Andersen, B. 1992a: Fine-grained parallelism in Ellie. Journal of Object-Oriented Progamming, 5(3):55-61. June 192.

Andersen, B. 1992b: Load balancing in fine-grained object-oriented language Ellie. in Cabrera, L.-F., Jul, E. (Eds.): Proceedings of the Second International Workshop on Object Orientation in Operating Systems. September 1992. IEEE Proceedings.

Andersen, B., 1993: On Ellie Object Granalarities. ECOOP´93. Workshop on Granularity of Objects in Distributed Systems. Position Paper. June 1993.

André, F., Pazat, J.-L., Thomas, H., 1990: Pandore: A system to manage datadistribution. Proceedings of the International Conference on Supercomputers. ACM. New York.

Bodin, F., Beckman, P., Gannon, D., Yang, S., Kesavan, S., Malony, A., Mohr, B., 1993: Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. Proceedings of the Supercomputing ´93 Conference. Portland, Oregon. November 15-19.

Booch, G., 1991: Object Oriented Design. Benjamin/Cummings, New York.

Carlin, P., Chandy, K., Kesselman, C., 1992: The Compositional C++ language definition. Technical Report CS-TR-92-02. Computer Science Department, California Institute of Technology.

Chien, A., Dally, W., 1990: Concurrent Aggregates (CA). ACM Sigplan Notices 25(3). March 1990.

Chien, A., 1993: Concurrent Aggregates. Supporting Modularity in Massively Parallel Programs. MIT Press, Cambridge.

Dally, W., et al, 1989: The J-Machine: A Fine-Grain Concurrent Computer. in: Proceedings of the IFIPS Conference.

Giloi, W., 1993: Rechnerarchitekturen. Springer-Verlag. Berlin.

Grimshaw, A., 1992: Easy-to-Use Object-Oriented Parallel Processing with Mentat. Technical Report No. CS-92-32. University of Virginia.

Grimshaw, A., 1993a: Easy-to-Use Object-Oriented Parallel Processing with Mentat. IEEE Computer, May 1993.

Grimshaw, A., Strayer, W.S., Narayan, P., 1993b: Dynamic, Object-Orientied Parallel Processing. IEEE Parallel & Distributed Technology, May 1993.

Hewitt, C., 1977: Viewing Control Structures as Pattern of Passing Messages. Artifical Intelligence, 8(3). Seite 323 - 364.

HPF Forum, 1993: Draft High Performance Fortran Language Spezification. Version 1.0. Erhältlich von titan.cs.rice.edu mittels ftp.

Jézéquel, J.-M., 1993: EPEE: An Eiffel environment to program distributed memory parallel computers. JOOP 6.2. May 1993.

Kale, L., Krishnan, S., 1993: CHARM++: A Portable Concurrent Object Oriented System Based On C++. ACM-Press. OOPSLA 93

Lieberman, H., 1986: Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. in Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. Portland, USA. Seite 214 - 223. November 1986.

Meyer, B., 1988: Object-oriented Software Construction. Prentice Hall.

Sivilotti, P., Carlin, P., 1994a: A Tutorial for CC++. Technical Report: Caltech CS-TR-94-02. Erhältlich von ftp.cs.caltech.edu mittels ftp.

Sivilotti, P., 1994b: A Verified Integration of Parallel Programming Paradigms in CC++. Sigel, E.(Ed.): 8th International Parallel Processing Symposium. April 26 - 29, 1994. Cancun, Mexico. IEEE Computer Society Press.

Yonezawa, A., Briot, J.-P., Shibayama, E., 1986: Object-oriented concurrent programming in ABCL/1. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. October 1986. ACM SIGPLAN Notices, 21(11): 258 - 268, November 1986.

Yonezawa, A., Shibayama, E., Honda, Y., Takada, T., Briot, J.-P., 1990: An Object-Oriented Concurrent Computing Model ABCM/1 and its Description Language ABCL/1. in Yonezawa, A. (Ed): ABCL. An Object-Oriented Concurrent System. MIT-Press, Cambridge.

Yonezawa, A., Matsuoka, S., Yasugi, M., Taura, K., 1993: Implementing Concurrent Object-Oriented Languages on Multicomputers. IEEE Parallel & Distributed Technology, May 1993.